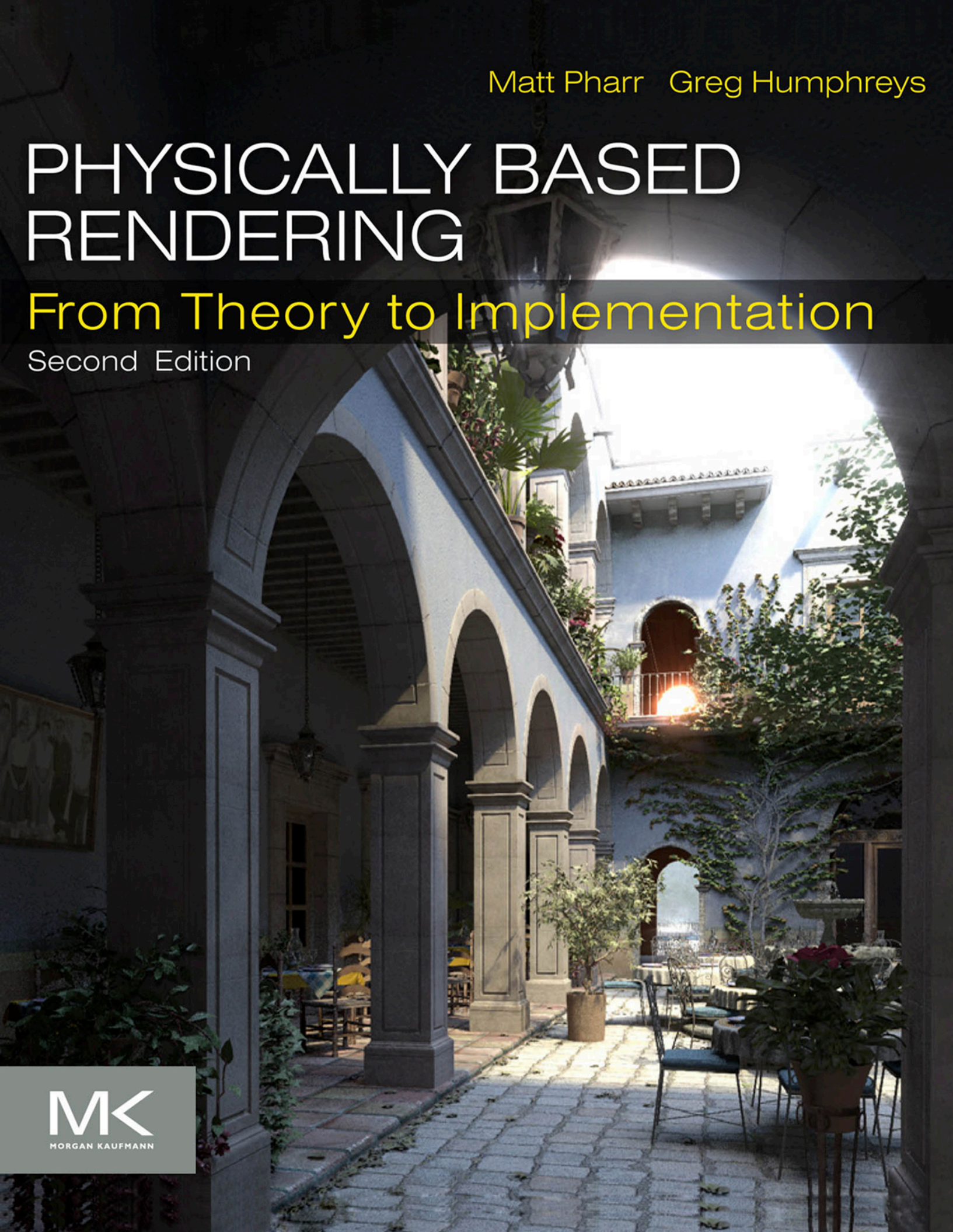
Matt Pharr Greg Humphreys

PHYSICALLY BASED RENDERING

From Theory to Implementation

Second Edition



MK
MORGAN KAUFMANN

Physically Based Rendering is a terrific book. It covers all the marvelous math, fascinating physics, practical software engineering, and clever tricks that are necessary to write a state-of-the-art photorealistic renderer. All of these topics are dealt with in a clear and pedagogical manner without omitting the all-important practical details.

`pbrt` *is not just a "toy" implementation of a ray tracer, but a general and robust full-scale global illumination renderer. It contains many important optimizations to reduce execution time and memory consumption for complex scenes. Furthermore,* `pbrt` *is easy to extend to experiment with other rendering algorithm variations.*

This book is not only a textbook for students, but also a useful reference book for practitioners in the field. The second edition has been extended with sections on Metropolis light transport, subsurface scattering, precomputed light transport, and more.

Per Christensen
Senior Software Developer, RenderMan Products, Pixar Animation Studios

Looking for a job in research or high end rendering? Get your kick-start education and start your own project with this book that comes along with both theory and real examples, meaning real code.

With their second edition, Matt Pharr and Greg Humphreys provide easy access to even the most advanced rendering techniques like Metropolis light transport and quasi-Monte Carlo methods. In addition the framework lets you skip the bootstrap pain of getting data into and out of your renderer.

The holistic approach of literate programming results in a clear logic of an easy-to-read text. If you are serious about graphics, there is no way around this unique and extremely valuable book that is closest to the state of the art.

Alexander Keller
Chief Scientist, Mental Images

This page intentionally left blank

Physically Based Rendering

FROM THEORY TO IMPLEMENTATION

SECOND EDITION

MATT PHARR
Intel

GREG HUMPHREYS
NVIDIA



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Acquiring Editor: Todd Green
Development Editor: Heather Scherer
Project Manager: Paul Gottehrer
Designer: Joanne Blank
Compositor: Paul C. Anagnostopoulos, Windfall Software

Morgan Kaufmann is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

© 2010 Elsevier, Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.
Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Application submitted

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-375079-2

Printed in China
10 11 12 13 14 10 9 8 7 6 5 4 3 2 1

Working together to grow
libraries in developing countries
www.elsevier.com | www.bookaid.org | www.sabre.org
ELSEVIER BOOK AID International Sabre Foundation

For information on all MK publications visit our website at www.mkp.com

To Deirdre, who even let me bring the manuscript on our honeymoon.

M. P.

To Isabel and Leila, the two most extraordinary people I've ever met. May your pixels never be little squares.

G. H.

ABOUT THE AUTHORS

Matt Pharr is a Principal Engineer at Intel, working as the lead architect in the Advanced Rendering Technology group. He previously co-founded Neoptica, which worked on programming models for graphics on heterogeneous CPU+GPU systems; Neoptica was acquired by Intel. Before Neoptica, Matt was in the Software Architecture group at NVIDIA, co-founded Exluna, and worked in Pixar's Rendering R&D group. He received his Ph.D. from the Stanford Graphics Lab, working under the supervision of Pat Hanrahan. He was also the editor of *GPU Gems 2*.

Greg Humphreys is a member of the OptiX raytracing team at NVIDIA. Previously, he was a professor of Computer Science at the University of Virginia, where he conducted research in both high performance and physically based computer graphics, as well as computer architecture and visualization. Greg has a B.S.E. degree from Princeton, and a Ph.D. in Computer Science from Stanford under the supervision of Pat Hanrahan. When he's not tracing rays, Greg enjoys tournament bridge, knitting, and riding his motorcycle.

Contents

PREFACE xix

CHAPTER 01. INTRODUCTION 1

1.1 Literate Programming 1
1.1.1 Indexing and Cross-Referencing 3
1.2 Photorealistic Rendering and the Ray-Tracing Algorithm 4
1.2.1 Cameras 5
1.2.2 Ray-Object Intersections 7
1.2.3 Light Distribution 8
1.2.4 Visibility 9
1.2.5 Surface Scattering 10
1.2.6 Recursive Ray Tracing 11
1.2.7 Ray Propagation 13
1.3 pbrt: System Overview 15
1.3.1 Phases of Execution 16
1.3.2 Scene Representation 18
1.3.3 Renderer Interface and `SamplerRenderer` 24
1.3.4 The Main Rendering Loop 26
1.3.5 Parallelization of `pbrt` 35
1.3.6 An Integrator for Whitted Ray Tracing 41
1.4 How to Proceed through This Book 47
1.4.1 The Exercises 48
1.5 Using and Understanding the Code 48
1.5.1 Pointer or Reference? 48
1.5.2 Code Optimization 49
1.5.3 The Book Web site 49
1.5.4 Extending the System 49
1.5.5 Bugs 50
Further Reading 50
Exercise 52

CHAPTER 02. GEOMETRY AND TRANSFORMATIONS 55

2.1 Coordinate Systems 55
2.1.1 Coordinate System Handedness 56
2.2 Vectors 57
2.2.1 Arithmetic 58
2.2.2 Scaling 59
2.2.3 Dot and Cross Product 60

2.2.4 Normalization 62
2.2.5 Coordinate System from a Vector 63

2.3 Points 63

2.4 Normals 65

2.5 Rays 66

2.5.1 Ray Differentials 68

2.6 Three-Dimensional Bounding Boxes 70

2.7 Transformations 74

2.7.1 Homogeneous Coordinates 75
2.7.2 Basic Operations 76
2.7.3 Translations 77
2.7.4 Scaling 79
2.7.5 x, y, and z Axis Rotations 80
2.7.6 Rotation around an Arbitrary Axis 82
2.7.7 The Look-At Transformation 84

2.8 Applying Transformations 85

2.8.1 Points 85
2.8.2 Vectors 86
2.8.3 Normals 86
2.8.4 Rays 88
2.8.5 Bounding Boxes 88
2.8.6 Composition of Transformations 88
2.8.7 Transformations and Coordinate System Handedness 89

*** 2.9 Animating Transformations 90**

2.9.1 Quaternions 92
2.9.2 Quaternion Interpolation 94
2.9.3 `AnimatedTransform` Implementation 96

2.10 Differential Geometry 101

Further Reading 103

Exercises 104

CHAPTER 03. SHAPES 107

3.1 Basic Shape Interface 107

3.1.1 Bounding 109
3.1.2 Refinement 110
3.1.3 Intersection 110
3.1.4 Avoiding Incorrect Self-Intersections 111
3.1.5 Shading Geometry 112
3.1.6 Surface Area 113
3.1.7 Sidedness 113

3.2 Spheres 113

3.2.1 Construction 115
3.2.2 Bounding 116

* An asterisk denotes a section with advanced content that can be skipped on a first reading.

3.2.3 Intersection 116
3.2.4 Partial Spheres 119
* 3.2.5 Partial Derivatives of Normal Vectors 121
3.2.6 `DifferentialGeometry` Initialization 122
3.2.7 Surface Area 123

3.3 Cylinders 124
3.3.1 Construction 125
3.3.2 Bounding 126
3.3.3 Intersection 126
3.3.4 Partial Cylinders 127
3.3.5 Surface Area 129

3.4 Disks 129
3.4.1 Construction 130
3.4.2 Bounding 130
3.4.3 Intersection 131
3.4.4 Surface Area 132

3.5 Other Quadrics 133
3.5.1 Cones 133
3.5.2 Paraboloids 134
3.5.3 Hyperboloids 134

3.6 Triangles and Meshes 135
3.6.1 Triangle 139
3.6.2 Triangle Intersection 140
3.6.3 Surface Area 145
3.6.4 Shading Geometry 145

*** 3.7 Subdivision Surfaces 148**
3.7.1 Mesh Representation 151
3.7.2 Bounds 160
3.7.3 Subdivison 160

Further Reading 174

Exercises 175

CHAPTER 04. PRIMITIVES AND INTERSECTION ACCELERATION 183

4.1 Primitive Interface and Geometric Primitives 184
4.1.1 Geometric Primitives 187
4.1.2 `TransformedPrimitive`: Object Instancing and Animated Primitives 189

4.2 Aggregates 191
4.2.1 Ray–Box Intersections 193

4.3 Grid Accelerator 195
4.3.1 Creation 197
4.3.2 Traversal 202

4.4 Bounding Volume Hierarchies 208
4.4.1 BVH Construction 210
4.4.2 The Surface Area Heuristic 217

4.4.3 Compact BVH For Traversal 222
4.4.4 Traversal 224
4.5 Kd-Tree Accelerator 227
4.5.1 Tree Representation 229
4.5.2 Tree Construction 231
4.5.3 Traversal 240
4.6 Debugging Aggregates 245
4.6.1 Finding Bugs in Aggregates 246
4.6.2 Fixing Bugs In Aggregates 249
4.6.3 Aggregate Performance Bugs 250
Further Reading 250
Exercises 255

CHAPTER 05. COLOR AND RADIOMETRY 261
5.1 Spectral Representation 261
5.1.1 The `Spectrum` Type 263
5.1.2 `CoefficientSpectrum` Implementation 263
5.2 The SampledSpectrum Class 266
5.2.1 XYZ Color 270
5.2.2 RGB Color 273
5.3 RGBSpectrum Implementation 279
5.4 Basic Radiometry 281
5.4.1 Basic Quantities 282
5.4.2 Incident and Exitant Radiance Functions 286
5.4.3 Luminance and photometry 287
5.5 Working with Radiometric Integrals 288
5.5.1 Integrals over Projected Solid Angle 289
5.5.2 Integrals over Spherical Coordinates 290
5.5.3 Integrals over Area 292
5.6 Surface Reflection 293
5.6.1 The BRDF 294
5.6.2 The BSSRDF 296
Further Reading 297
Exercises 298

CHAPTER 06. CAMERA MODELS 301
6.1 Camera Model 301
6.1.1 Camera Coordinate Spaces 303
6.2 Projective Camera Models 305
6.2.1 Orthographic Camera 306
6.2.2 Perspective Camera 310
6.2.3 Depth of Field 313
6.3 Environment Camera 318

Further Reading 320
Exercises 320

CHAPTER 07. SAMPLING AND RECONSTRUCTION 323

7.1 **Sampling Theory** 323
7.1.1 The Frequency Domain and the Fourier Transform 325
7.1.2 Ideal Sampling and Reconstruction 327
7.1.3 Aliasing 331
7.1.4 Antialiasing Techniques 332
7.1.5 Application to Image Synthesis 335
7.1.6 Sources of Aliasing in Rendering 336
7.1.7 Understanding Pixels 337
7.2 **Image Sampling Interface** 338
7.2.1 Sample Representation and Allocation 342
7.3 **Stratified Sampling** 346
* 7.4 **Low-Discrepancy Sampling** 359
7.4.1 Definition of Discrepancy 359
7.4.2 Hammersley and Halton Sequences 361
7.4.3 (0,2)-Sequences 368
7.4.4 The Low-Discrepancy Sampler 372
* 7.5 **Best-Candidate Sampling Patterns** 378
7.6 **Adaptive Sampling** 385
7.7 **Image Reconstruction** 389
7.7.1 Filter Functions 393
7.8 **Film and the Imaging Pipeline** 402
7.8.1 Film Interface 403
7.8.2 ImageFilm 404
Further Reading 413
Exercises 417

CHAPTER 08. REFLECTION MODELS 423

8.1 **Basic Interface** 428
8.1.1 Reflectance 430
8.1.2 BRDF → BTDF Adapter 430
8.1.3 BxDF Scaling Adapter 431
8.2 **Specular Reflection and Transmission** 432
8.2.1 Fresnel Reflectance 434
8.2.2 Specular Reflection 439
8.2.3 Specular Transmission 442
8.3 **Lambertian Reflection** 446
8.4 **Microfacet Models** 447
8.4.1 Oren-Nayar Diffuse Reflection 449
8.4.2 Torrance-Sparrow Model 452

8.4.3 Blinn Microfacet Distribution 455
8.4.4 Anisotropic Microfacet Model 457
8.5 Fresnel Incidence Effects 460
8.6 Measured BRDFs 462
8.6.1 Irregular Isotropic Measured BRDF 463
8.6.2 Regular Halfangle Format 467
Further Reading 470
Exercises 472

CHAPTER 09. MATERIALS 477
9.1 BSDFs 477
9.1.1 BSDF Memory Management 482
9.2 Material Interface and Implementations 483
9.2.1 MatteMaterial 484
9.2.2 PlasticMaterial 486
9.2.3 Mix Material 487
9.2.4 Measured Material 489
9.2.5 Additional Materials 490
9.3 Bump Mapping 490
Further Reading 496
Exercises 497

CHAPTER 10. TEXTURE 501
10.1 Sampling and Antialiasing 503
10.1.1 Finding the Texture Sampling Rate 503
10.1.2 Filtering Texture Functions 509
* 10.1.3 Ray Differentials for Specular Reflection and Transmission 510
10.2 Texture Coordinate Generation 513
10.2.1 2D (u, v) Mapping 514
10.2.2 Spherical Mapping 515
10.2.3 Cylindrical Mapping 517
10.2.4 Planar Mapping 518
10.2.5 3D Mapping 518
10.3 Texture Interface and Basic Textures 519
10.3.1 Constant Texture 520
10.3.2 Scale Texture 520
10.3.3 Mix Textures 521
10.3.4 Bilinear Interpolation 522
10.4 Image Texture 523
10.4.1 Texture Caching 524
10.4.2 MIP Maps 528
10.4.3 Isotropic Triangle Filter 536
* 10.4.4 Elliptically Weighted Average 539
10.5 Solid and Procedural Texturing 544
10.5.1 UV Texture 545

10.5.2 Checkerboard 546
10.5.3 Solid Checkerboard 552
10.6 Noise 553
10.6.1 Perlin Noise 554
10.6.2 Random Polka Dots 558
10.6.3 Noise Idioms and Spectral Synthesis 560
10.6.4 Bumpy and Wrinkled Textures 565
10.6.5 Windy Waves 566
10.6.6 Marble 567
Further Reading 569
Exercises 571

* CHAPTER 11. VOLUME SCATTERING 575
11.1 Volume Scattering Processes 575
11.1.1 Absorption 576
11.1.2 Emission 578
11.1.3 Out-Scattering and Extinction 579
11.1.4 In-scattering 581
11.2 Phase Functions 583
11.3 Volume Interface and Homogeneous Media 587
11.3.1 Homogeneous Volumes 589
11.4 Varying-Density Volumes 591
11.4.1 3D Grids 592
11.4.2 Exponential Density 594
11.5 Volume Aggregates 596
11.6 The BSSRDF 598
11.6.1 Subsurface Scattering Materials 599
Further Reading 600
Exercises 601

CHAPTER 12. LIGHT SOURCES 605
12.1 Light Interface 606
12.1.1 Visibility Testing 608
12.2 Point Lights 609
12.2.1 Spotlights 611
12.2.2 Texture Projection Lights 614
12.2.3 Goniophotometric Diagram Lights 618
12.3 Distant Lights 621
12.4 Area Lights 623
12.5 Infinite Area Lights 627
Further Reading 632
Exercises 633

CHAPTER 13. MONTE CARLO INTEGRATION I: BASIC CONCEPTS 637

13.1 **Background and Probability Review** 638
13.1.1 Continuous Random Variables 639
13.1.2 Expected Values and Variance 640

13.2 **The Monte Carlo Estimator** 641

13.3 **Basic Sampling Of Random Variables** 643
13.3.1 The Inversion Method 643
13.3.2 The Rejection Method 650

* 13.4 **Metropolis Sampling** 652
13.4.1 Basic Algorithm 652
13.4.2 Choosing Mutation Strategies 654
13.4.3 Start-up bias 655
13.4.4 Estimating Integrals With Metropolis Sampling 656
13.4.5 Example: One-Dimensional Setting 656

13.5 **Transforming between Distributions** 660
13.5.1 Transformation in Multiple Dimensions 661
13.5.2 Example: Polar Coordinates 661
13.5.3 Example: Spherical Coordinates 661

13.6 **2D Sampling with Multidimensional Transformations** 662
13.6.1 Example: Uniformly Sampling a Hemisphere 663
13.6.2 Example: Sampling a Unit Disk 665
13.6.3 Example: Cosine-Weighted Hemisphere Sampling 668
13.6.4 Example: Sampling a Triangle 670
13.6.5 Example: Piecewise-Constant 2D Distributions 671

Further Reading 674

Exercises 675

CHAPTER 14. MONTE CARLO INTEGRATION II: IMPROVING EFFICIENCY 679

14.1 **Russian Roulette and Splitting** 680
14.1.1 Splitting 681

14.2 **Careful Sample Placement** 682
14.2.1 Stratified Sampling 682
14.2.2 Quasi Monte Carlo 685
14.2.3 Warping Samples and Distortion 685

14.3 **Bias** 686

14.4 **Importance Sampling** 688
14.4.1 Multiple Importance Sampling 690

14.5 **Sampling Reflection Functions** 693
14.5.1 Sampling the Blinn Microfacet Distribution 695
14.5.2 Sampling the Anisotropic Microfacet Model 699
14.5.3 Sampling `FresnelBlend` 701
14.5.4 Specular Reflection and Transmission 702

14.5.5 Application: Estimating Reflectance 703
14.5.6 Sampling BSDFs 705
14.6 Sampling Light Sources 708
14.6.1 Basic Interface 709
14.6.2 Lights with Singularities 711
14.6.3 Area Lights 715
14.6.4 ShapeSet Sampling 722
14.6.5 Infinite Area Lights 724
* 14.7 Volume Scattering 730
14.7.1 Sampling Phase Functions 731
14.7.2 Computing Optical Thickness 731
Further Reading 733
Exercises 734

CHAPTER 15. LIGHT TRANSPORT I: SURFACE REFLECTION 739
15.1 Direct Lighting 741
15.1.1 Estimating the Direct Lighting Integral 747
15.2 The Light Transport Equation 751
15.2.1 Basic Derivation 751
15.2.2 Analytic Solutions to the LTE 753
15.2.3 The Surface Form of the LTE 754
15.2.4 Integral over Paths 755
15.2.5 Delta distributions in the Integrand 757
15.2.6 Partitioning the Integrand 758
15.2.7 The Measurement Equation and Importance 759
15.3 Path Tracing 760
15.3.1 Overview 762
15.3.2 Path Sampling 763
15.3.3 Incremental Path Construction 765
15.3.4 Implementation 765
* 15.3.5 Bidirectional Path Tracing 770
15.4 Instant Global Illumination 772
15.4.1 Creating the virtual light sources 775
15.4.2 Rendering with virtual light sources 780
15.5 Irradiance Caching 784
15.5.1 Rendering With The Irradiance Cache 789
15.5.2 Lookup and Interpolatiton 791
15.5.3 Adding new values 794
15.6 Particle Tracing and Photon Mapping 797
* 15.6.1 Theoretical Basis for Particle Tracing 798
15.6.2 Photon Integrator 800
15.6.3 Building the Photon Maps 803
15.6.4 Using the Photon Map 817
15.6.5 Photon Interpolation and Density Estimation 827

* 15.7 Metropolis Light Transport 833
15.7.1 Sample Representation 838
15.7.2 Mutations 839
15.7.3 Generating Paths 842
15.7.4 Path Contributions 845
15.7.5 `MetropolisRenderer` Implementation 852
15.7.6 Rendering 857
Further Reading 861
Exercises 865

* CHAPTER 16. LIGHT TRANSPORT II: VOLUME RENDERING 873
16.1 The Equation of Transfer 873
16.2 Volume Integrator Interface 876
16.3 Emission-Only Integrator 876
16.4 Single Scattering Integrator 882
16.5 Subsurface Scattering 885
16.5.1 Poisson Distribution of Sample Points 888
16.5.2 Building the Sample Point Octree 897
16.5.3 The Dipole Diffusion Approximation 901
16.5.4 Rendering with Hierarchical Integration 908
16.5.5 Setting Scattering Properties 912
Further Reading 913
Exercises 916

* CHAPTER 17. LIGHT TRANSPORT III: PRECOMPUTED LIGHT TRANSPORT 925
17.1 Basis Functions: Theory 927
17.1.1 A Piecewise-Constant Basis 928
17.1.2 Projection Onto a Basis 930
17.1.3 Orthonormal Basis Functions 930
17.2 Spherical Harmonics 932
17.2.1 Efficient Evaluation 935
17.2.2 Projecting Light Sources 941
17.2.3 Projecting Incident Radiance Functions 949
17.2.4 Reducing Ringing 950
17.2.5 Rotations 951
17.3 Radiance Probes 956
17.3.1 Creating Radiance Probes 956
17.3.2 Using Radiance Probes 965
17.4 Precomputed Diffuse Transfer 969
17.5 Precomputed Glossy Transfer 974
17.5.1 The Transfer Matrix 975

17.5.2 The BSDF Matrix 977
17.5.3 `GlossyPRTIntegrator` Implementation 979
Further Reading **982**
Exercises **984**

CHAPTER 18. RETROSPECTIVE AND THE FUTURE **989**

18.1 Design Retrospective **989**
18.1.1 Abstraction versus Efficiency 991
18.1.2 Design Alternative: Triangles Only 991
18.1.3 Increased Scene Complexity 992
18.2 Throughput Processors **993**
18.2.1 The Future 995
18.2.2 Further Resources 995
18.3 Conclusion **997**

APPENDIXES

A UTILITIES **999**
B SCENE DESCRIPTION INTERFACE **1045**
C INDEX OF FRAGMENTS **1077**
D INDEX OF CLASSES AND THEIR MEMBERS **1091**
E INDEX OF MISCELLANEOUS IDENTIFIERS **1101**

REFERENCES 1105
INDEX 1143
COLOPHON 1169

This page intentionally left blank

Preface

[Just as] other information should be available to those who want to learn and understand, program source code is the only means for programmers to learn the art from their predecessors. It would be unthinkable for playwrights not to allow other playwrights to read their plays [or to allow them] at theater performances where they would be barred even from taking notes. Likewise, any good author is well read, as every child who learns to write will read hundreds of times more than it writes. Programmers, however, are expected to invent the alphabet and learn to write long novels all on their own. Programming cannot grow and learn unless the next generation of programmers has access to the knowledge and information gathered by other programmers before them. —Erik Naggum

Rendering is a fundamental component of computer graphics. At the highest level of abstraction, rendering is the process of converting a description of a three-dimensional scene into an image. Algorithms for animation, geometric modeling, texturing, and other areas of computer graphics all must pass their results through some sort of rendering process so that they can be made visible in an image. Rendering has become ubiquitous; from movies to games and beyond, it has opened new frontiers for creative expression, entertainment, and visualization.

In the early years of the field, research in rendering focused on solving fundamental problems such as determining which objects are visible from a given viewpoint. As effective solutions to these problems have been found and as richer and more realistic scene descriptions have become available thanks to continued progress in other areas of graphics, modern rendering has grown to include ideas from a broad range of disciplines, including physics and astrophysics, astronomy, biology, psychology and the study of perception, and pure and applied mathematics. The interdisciplinary nature of rendering is one of the reasons that it is such a fascinating area of study.

This book presents a selection of modern rendering algorithms through the documented source code for a complete rendering system. All of the images in this book, including the one on the front cover, were rendered by this software. All of the algorithms that came together to generate these images are described in these pages. The system, `pbrt`, is written using a programming methodology called *literate programming* that mixes prose describing the system with the source code that implements it. We believe that the literate programming approach is a valuable way to introduce ideas in computer graphics and computer science in general. Often, some of the subtleties of an algorithm can be unclear or hidden until it is implemented, so seeing an actual implementation is a good way to acquire a solid understanding of that algorithm's details. Indeed, we believe that deep understanding of a small number of algorithms in this manner provides a stronger base for further study of computer graphics than does superficial understanding of many.

In addition to clarifying how an algorithm is implemented in practice, presenting these algorithms in the context of a complete and nontrivial software system also allows us to address issues in the design and implementation of medium-sized rendering systems. The design of a rendering system's basic abstractions and interfaces has substantial implications for both the elegance of the implementation and the ability to extend it later, yet the trade-offs in this design space are rarely discussed.

pbrt and the contents of this book focus exclusively on *photorealistic rendering*, which can be defined variously as the task of generating images that are indistinguishable from those that a camera would capture in a photograph, or as the task of generating images that evoke the same response from a human observer as looking at the actual scene. There are many reasons to focus on photorealism. Photorealistic images are crucial for the movie special-effects industry because computer-generated imagery must often be mixed seamlessly with footage of the real world. In entertainment applications where all of the imagery is synthetic, photorealism is an effective tool for making the observer forget that he or she is looking at an environment that does not actually exist. Finally, photorealism gives a reasonably well-defined metric for evaluating the quality of the rendering system's output.

A consequence of our approach is that this book and the system it describes do not exhaustively cover the state-of-the-art in rendering; many interesting topics in photorealistic rendering will not be introduced either because they don't fit well with the architecture of the software system (e.g., finite-element radiosity algorithms) or because we believed that the pedagogical value of explaining the algorithm was outweighed by the complexity of its implementation. We will note these decisions as they come up and provide pointers to further resources so that the reader can follow up on topics of interest. Many other areas of rendering, including interactive rendering, visualization, and illustrative forms of rendering such as pen-and-ink styles, aren't covered in this book at all. Nevertheless, many of the algorithms and ideas in this system (e.g., algorithms for texture map antialiasing) are applicable to a wider set of rendering styles.

AUDIENCE

Our primary intended audience for this book is students in graduate or upper-level undergraduate computer graphics classes. This book assumes existing knowledge of computer graphics at the level of an introductory college-level course, although certain key concepts such as basic vector geometry and transformations will be reviewed here. For students who do not have experience with programs that have tens of thousands of lines of source code, the literate programming style gives a gentle introduction to this complexity. We pay special attention to explaining the reasoning behind some of the key interfaces and abstractions in the system in order to give these readers a sense of why the system is structured in the way that it is.

Our secondary, but equally important, audiences are advanced graduate students and researchers, software developers in industry, and individuals interested in the fun of writing their own rendering systems. Although many of the ideas in this book will likely be familiar to these readers, seeing explanations of the algorithms presented in the literate style may provide new perspectives. pbrt includes implementations of a number of

advanced and/or difficult-to-implement algorithms and techniques, such as subdivision surfaces, Monte Carlo light transport and Metropolis sampling, subsurface scattering, and precomputed light transport algorithms; these should be of particular interest to experienced practitioners in rendering. We hope that delving into one particular organization of a complete and nontrivial rendering system will also be thought provoking to this audience.

OVERVIEW AND GOALS

`pbrt` is based on the *ray-tracing* algorithm. Ray tracing is an elegant technique that has its origins in lens making; Carl Freidrich Gauss traced rays through lenses by hand in the 19^{th} century. Ray-tracing algorithms on computers follow the path of infinitesimal rays of light through the scene until they intersect a surface. This approach gives a simple method for finding the first visible object as seen from any particular position and direction, and is the basis for many rendering algorithms.

`pbrt` was designed and implemented with three main goals in mind: it should be *complete*, it should be *illustrative*, and it should be *physically based*.

Completeness implies that the system should not lack key features found in high-quality commercial rendering systems. In particular, it means that important practical issues, such as antialiasing, robustness, and the ability to efficiently render complex scenes, should all be addressed thoroughly. It is important to consider these issues from the start of the system's design, since these features can have subtle implications for all components of the system and can be quite difficult to retrofit into the system at a later stage of implementation.

Our second goal means that we tried to choose algorithms, data structures, and rendering techniques with care and with an eye toward readability and clarity. Since their implementations will be examined by more readers than is the case for many other rendering systems, we tried to select the most elegant algorithms that we were aware of and implement them as well as possible. This goal also required that the system be small enough for a single person to understand completely. We have implemented `pbrt` using an extensible architecture, with the core of the system implemented in terms of a set of carefully-designed abstract base classes, and as much of the specific functionality as possible in implementations of these base classes. The result is that one doesn't need to understand all of the specific implementations in order to understand the basic structure of the system. This makes it easier to delve deeply into parts of interest and skip others, without losing sight of how the overall system fits together.

There is a tension between the two goals of being complete and being illustrative. Implementing and describing every possible useful technique would not only make this book extremely long, but also would make the system prohibitively complex for most readers. In cases where `pbrt` lacks a particularly useful feature, we have attempted to design the architecture so that the feature could be added without altering the overall system design.

The basic foundations for physically based rendering are the laws of physics and their mathematical expression. `pbrt` was designed to use the correct physical units and concepts for the quantities it computes and the algorithms it implements. When configured

to do so, pbrt can compute images that are *physically correct*; they accurately reflect the lighting as it would be in a real-world version of the scene. One advantage of the decision to use a physical basis is that it gives a concrete standard of program correctness: for simple scenes, where the expected result can be computed in closed form, if pbrt doesn't compute the same result, we know there must be a bug in the implementation. Similarly, if different physically based lighting algorithms in pbrt give different results for the same scene, or if pbrt doesn't give the same results as another physically based renderer, there is certainly an error in one of them. Finally, we believe that this physically based approach to rendering is valuable because it is rigorous. When it is not clear how a particular computation should be performed, physics gives an answer that guarantees a consistent result.

Efficiency was given lower priority than these three goals. Since rendering systems often run for many minutes or hours in the course of generating an image, efficiency is clearly important. However, we have mostly confined ourselves to *algorithmic* efficiency rather than low-level code optimization. In some cases, obvious micro-optimizations take a backseat to clear, well-organized code, although we did make some effort to optimize the parts of the system where most of the computation occurs.

In the course of presenting pbrt and discussing its implementation, we hope to convey some hard-learned lessons from years of rendering research and development. There is more to writing a good renderer than stringing together a set of fast algorithms; making the system both flexible and robust is a difficult task. The system's performance must degrade gracefully as more geometry or light sources are added to it, or as any other axis of complexity is pushed. Numeric stability must be handled carefully, and algorithms that don't waste floating-point precision are critical.

The rewards for developing a system that addresses all these issues are enormous—it is a great pleasure to write a new renderer or add a new feature to an existing renderer and use it to create an image that couldn't be generated before. Our most fundamental goal in writing this book was to bring this opportunity to a wider audience. Readers are encouraged to use the system to render the example scenes in the pbrt software distribution as they progress through the book. Exercises at the end of each chapter suggest modifications to the system that will help clarify its inner workings, and more complex projects to extend the system by adding new features.

The website for this book is located at *www.pbrt.org*. The latest version of the pbrt source code is available from this site and we will also post errata and bug fixes, additional scenes to render, and supplemental utilities. Any bugs in pbrt or errors in this text that are not listed at the website can be reported to the email address *bugs@pbrt.org*. We greatly value your feedback!

CHANGES SINCE THE FIRST EDITION

Six years have passed since the publication of the first edition of this book. In that time, thousands of copies of the book have been sold and the pbrt software has been downloaded thousands of times from the book's website. The pbrt user base has given

us a significant amount of feedback and encouragement, and our experience with the system guided many of the decisions we made in making changes between the version of pbrt presented in the first edition and the updated version described here. In addition to a number of bug fixes, we also made several significant design changes and enhancements:

1. Removal of the plugin architecture. The first version of pbrt used a run-time plugin architecture to dynamically load code for implementations of objects like shapes, lights, integrators, cameras, and other objects that were used in the scene currently being rendered. This approach allowed users to extend pbrt with new object types (e.g., new shape primitives) without recompiling the entire rendering system. This approach initially seemed elegant, but it complicated the task of supporting pbrt on multiple platforms and it made debugging more difficult. The only new usage scenario that it truly enabled (binary-only distributions of pbrt or binary plugins) was actually contrary to our pedagogical and open-source goals. Therefore, the plugin architecture was dropped in this edition.
2. Removal of the image processing pipeline. The first version of pbrt provided a tone-mapping interface that converted high dynamic range (HDR) floating point output images directly into low dynamic range TIFFs for display. This functionality made sense in 2004, as support for HDR images was still sparse. In 2010, however, advances in digital photography have made HDR images commonplace. Although the theory and practice of tone mapping are elegant and worth learning, we decided to focus the new book exclusively on the process of image formation and ignore the topic of image display. Interested readers should read the book written by Reinhard et al. (2005) for a thorough and modern treatment of the HDR image display process.
3. Task parallelism. Multicore architectures are now ubiquitous, and we felt that pbrt would not remain relevant without the ability to scale to the number of locally available cores. We also hope that the parallel programming implementation details documented in this book will help graphics programmers understand some of the subtleties and complexities in writing scalable parallel code (e.g., choosing appropriate task granularities or mutex types), which is still a difficult and too-infrequently taught topic.
4. Appropriateness for "production" rendering. The first version of pbrt was intended exclusively as a pedagogical tool and a stepping-stone for rendering research. Indeed, we made a number of decisions in preparing the first edition that were contrary to use in a production environment, such as limited support for image-based lighting, no support for motion blur, and a photon mapping implementation that wasn't robust in the presence of complex lighting. With much improved support for these features as well as support for subsurface scattering and Metropolis light transport, we feel that pbrt is now much more suitable for rendering very high-quality images of complex environments as it is presented here. The tradeoff in making these improvements is that as the system becomes more feature-complete, it may be harder for instructors to use the new software to create manageable assignments for students. While this is a real concern, we had similar reservations about the first version of pbrt "relieving" students of the burden and benefits of writing their own ray-tracing system from scratch. With

experience from the first edition being used at many universities, we have come to believe that this tradeoff was a good one, and we hope and expect that the new edition will continue to enable high-quality rendering courses.

ACKNOWLEDGMENTS

Pat Hanrahan has contributed to this book in more ways than we could hope to acknowledge; we owe a profound debt to him. He tirelessly argued for clean interfaces and finding the right abstractions to use throughout the system, and his understanding of and approach to rendering deeply influenced its design. His willingness to use `pbrt` and this manuscript in his rendering course at Stanford was enormously helpful, particularly in the early years of its life when it was still in very rough form; his feedback throughout this process has been crucial for bringing the text to its current state. Finally, the group of people that Pat helped assemble at the Stanford Graphics Lab, and the open environment that he fostered, made for an exciting, stimulating, and fertile environment. We feel extremely privileged to have been there.

We owe a debt of gratitude to the many students who used early drafts of this book in courses at Stanford and the University of Virginia between 1999 and 2004. These students provided an enormous amount of feedback about the book and `pbrt`. The teaching assistants for these courses deserve special mention: Tim Purcell, Mike Cammarano, Ian Buck, and Ren Ng at Stanford, and Nolan Goodnight at Virginia. A number of students in those classes gave particularly valuable feedback and sent bug reports and bug fixes; we would especially like to thank Evan Parker and Phil Beatty. A draft of the manuscript of this book was used in classes taught by Bill Mark and Don Fussell at the University of Texas, Austin, and Raghu Machiraju at Ohio State University; their feedback was invaluable, and we are grateful for their adventurousness in incorporating this system into their courses, even while it was still being edited and revised.

Matt Pharr would like to acknowledge colleagues and co-workers in rendering-related endeavors who have been a great source of education and who have substantially influenced his approach to writing renderers and his understanding of the field. Particular thanks go to Craig Kolb, who provided a cornerstone of Matt's early computer graphics education through the freely available source code to the `rayshade` ray-tracing system, and Eric Veach, who has also been generous with his time and expertise. Thanks also to Doug Shult and Stan Eisenstat for formative lessons in mathematics and computer science during high school and college, respectively, and most importantly to Matt's parents, for the education they've provided and continued encouragement along the way. Finally, thanks also to Nick Triantos, Jayant Kolhe, and NVIDIA for their understanding and support through the final stages of the preparation of the first edition of the book.

Greg Humphreys is very grateful to all the professors and TAs who tolerated him when he was an undergraduate at Princeton. Many people encouraged his interest in graphics, specifically Michael Cohen, David Dobkin, Adam Finkelstein, Michael Cox, Gordon Stoll, Patrick Min, and Dan Wallach. Doug Clark, Steve Lyon, and Andy Wolfe also supervised various independent research boondoggles without even laughing once. Once, in a group meeting about a year-long robotics project, Steve Lyon became exasperated and yelled, "Stop telling me why it can't be done, and figure out how to do it!"—an im-

promptu lesson that will never be forgotten. Eric Ristad fired Greg as a summer research assistant after his freshman year (before the summer even began), pawning him off on an unsuspecting Pat Hanrahan and beginning an advising relationship that would span 10 years and both coasts. Finally, Dave Hanson taught Greg that literate programming was a great way to work, and that computer programming can be a beautiful and subtle art form.

We are also grateful to Don Mitchell, for his help with understanding some of the details of sampling and reconstruction; Thomas Kollig and Alexander Keller, for explaining the finer points of low-discrepancy sampling; and Dave Eberly, "Just d'FAQs," Hans-Bernhard Broeker, Steve Westin, and Gernot Hoffmann, for many interesting threads on *comp.graphics.algorithms*. Christer Ericson had a number of suggestions for improving our kd-tree implementation. Christophe Hery helped us with understanding the nuances of subsurface scattering and Peter-Pike Sloan was kind enough to carefully review Chapter 17 on precomputed light transport algorithms.

Many people and organizations have generously supplied us with scenes and models for use in this book and the `pbrt` distribution. Their generosity has been invaluable in helping us create interesting example images throughout the text. The bunny, Buddha, and dragon models are courtesy of the Stanford Computer Graphics Laboratory's scanning repository at *graphics.stanford.edu/data/3Dscanrep/*. The ecosystem scene was created by Oliver Deussen and Bernd Lintermann for a paper by them and collaborators (Deussen, Hanrahan, Lintermann, Mech, Pharr, and Prusinkiewicz 1998). The "killeroo" model is included with permission of Phil Dench and Martin Rezard (3D scan and digital representations by headus, design and clay sculpt by Rezard). The physically accurate smoke data sets were created by Duc Nguyen and Ron Fedkiw. Nolan Goodnight created environment maps with a realistic skylight model. The Cornell Program of Computer Graphics Light Measurement Laboratory allowed us to include measured BRDF data, and Paul Debevec provided numerous high dynamic-range environment maps. Marc Ellens provided spectral data for a variety of light sources, and the spectral RGB measurement data for a variety of displays is courtesy of Tom Lianza at X-Rite.

We are particularly grateful to Guillermo M. Leal Llaguno of Evolución Visual, *www.evvisual.com*, who modeled and rendered the San Miguel scene featured on the cover and in numerous figures in the book. We would also especially like to thank Marko Dabrovic (*www.3lhd.com*) and Mihovil Odak at RNA Studios (*www.rna.hr*), who supplied a bounty of excellent models and scenes, including the Sponza atrium, the Sibenik cathedral, and the Audi TT car model. Many thanks are also due to Florent Boyer (*www.florentboyer.com*), who provided the contemporary house scene used in many of the images in Chapter 15.

We would also like to thank the book's reviewers, all of whom had insightful and constructive feedback about the manuscript at various stages of its progress. We'd particularly like to thank the reviewers who provided feedback on both editions of the book: Ian Ashdown, Per Christensen, Doug Epps, Dan Goldman, Eric Haines, Erik Reinhard, Pete Shirley, Peter-Pike Sloan, Greg Ward, and a host of anonymous reviewers. For the second edition, Janne Kontkanen, Nelson Max, Bill Mark, and Eric Tabellion also contributed numerous helpful suggestions.

We'd like to thank the faculty members at various universities who have used pbrt in their courses, including Emmanuel Agu, Dirk Arnold, Stephen Chenney, Yung-Yu Chuang, Don Fussell, Pat Hanrahan, Bill Mark, Nelson Max, Gary Meyer, Torsten Möller, Rick Parent, Sumanta Pattanaik, and Luiz Velho.

Many people have contributed to not only pbrt but to our own better understanding of rendering through bug reports, patches, and suggestions about better implementation aproaches. A few have made particularly substantial contributions—we would especially like to thank Kevin Egan, John Danks, Volodymyr Kachurovskyi, Solomon Boulos, and Stephen Chenney. In addition, we would also like to thank Rachit Agrawal, Frederick Akalin, Mark Bolstad, Thomas de Bodt, Brian Budge, Mark Colbert, Shaohua Fan, Nigel Fisher, Jeppe Revall Frisvad, Robert G. Graf, Asbjørn Heid, Keith Jeffery, Greg Johnson, Aaron Karp, Donald Knuth, Martin Kraus, Murat Kurt, Larry Lai, Craig McNaughton, Swaminathan Narayanan, Anders Nilsson, Jens Olsson, Vincent Pegoraro, Nils Thuerey, Xiong Wei, Wei-Wei Xu, Arek Zimny, and Matthias Zwicker for their suggestions and bug reports. Finally, we would like to thank the *LuxRender* developers and the *LuxRender* community, particularly Terrence Vergauwen, Jean-Philippe Grimaldi, and Asbjørn Heid; it has been a delight to see the rendering system they have built from pbrt's foundation, and we have learned from reading their source code and implementations of new rendering algorithms.

For the production of the first edition, we would also like to thank Tim Cox (senior editor), for his willingness to take on this slightly unorthodox project and for both his direction and patience throughout the process. We are very grateful to Elisabeth Beller (project manager), who has gone well beyond the call of duty for this book; her ability to keep this complex project in control and on schedule has been remarkable, and we particularly thank her for the measurable impact she has had on the quality of the final result. Thanks also to Rick Camp (editorial assistant) for his many contributions along the way. Paul Anagnostopoulos and Jacqui Scarlott at Windfall Software did the book's composition; their ability to take the authors' homebrew literate programming file format and turn it into high-quality final output while also juggling the multiple unusual types of indexing we asked for is greatly appreciated. Thanks also to Ken DellaPenta (copyeditor) and Jennifer McClain (proofreader) as well as to Max Spector at Chen Design (text and cover designer), and Steve Rath (indexer).

For the second edition, we'd like to thank Greg Chalson who talked us into expanding and updating the book; Greg also ensured that Paul Anagnostopoulos at Windfall Software would again do the book's composition. We'd like to thank Paul again for his efforts in working with this book's production complexity. Finally, we'd also like to thank Todd Green, Paul Gottehrer, and Heather Scherer at Elsevier.

ABOUT THE COVER

The "San Miguel" scene on the cover of the book was modeled and then rendered by Guillermo M. Leal Llaguno of Evolución Visual, *www.evvisual.com*, based on a hacienda that he visited in San Miguel de Allende, Mexico. The scene was modeled in *3ds max* and exported to the pbrt file format with a custom script written by Guillermo. The scene features just over 2.5 million unique triangles and has a total geometric complexity of

10.7 million triangles due to the use of object instancing; the `pbrt` files that describe the scene geometry require 620 MB of on-disk storage. There are a total of 354 texture maps, representing 293 MB of texture data. Final rendering of the cover image at 1496 by 2235 resolution using `pbrt` took over 40 hours of computation on an eight-core Mac Pro computer. The scene is available in the `scenes/sanmiguel` directory of the `pbrt` distribution.

ADDITIONAL READING

Donald Knuth's article *Literate Programming* (Knuth 1984) describes the main ideas behind literate programming as well as his `web` programming environment. The seminal TEX typesetting system was written with `web` and has been published as a series of books (Knuth 1986, Knuth 1993a). More recently, Knuth has published a collection of graph algorithms in literate format in *The Stanford GraphBase* (Knuth 1993b). These programs are enjoyable to read and are excellent presentations of their respective algorithms. The website *www.literateprogramming.com* has pointers to many articles about literate programming, literate programs to download, and a variety of literate programming systems; many refinements have been made since Knuth's original development of the idea.

The only other literate programs we know of that have been published as books are the implementation of the `lcc` compiler, which was written by Christopher Fraser and David Hanson and published as *A Retargetable C Compiler: Design and Implementation* (Fraser and Hanson 1995), and Martin Ruckert's book on the *mp3* audio format, *Understanding MP3* (Ruckert 2005).



CHAPTER ONE

01 INTRODUCTION

Rendering is the process of producing a 2D image from a description of a 3D scene. Obviously, this is a very broad task, and there are many ways to approach it. *Physically based* techniques attempt to simulate reality; that is, they use principles of physics to model the interaction of light and matter. In physically based rendering, realism is usually the primary goal. This approach is in contrast to *interactive* rendering, which sacrifices realism for high performance and low latency, or *nonphotorealistic* rendering, which strives for artistic freedom and expressiveness.

This book describes `pbrt`, a physically based rendering system based on the ray-tracing algorithm. Most computer graphics books present algorithms and theory, sometimes combined with snippets of code. In contrast, this book couples the theory with a complete implementation of a fully functional rendering system. The source code to the system (as well as example scenes and a collection of data for rendering) can be found from the `pbrt` Web site's downloads page, *pbrt.org/downloads.php*.

1.1 LITERATE PROGRAMMING

While writing the TEX typesetting system, Donald Knuth developed a new programming methodology based on the simple but revolutionary idea that *programs should be written more for people's consumption than for computers' consumption*. He named this methodology *literate programming*. This book (including the chapter you're reading now) is a long literate program. This means that in the course of reading this book, you will read the *full* implementation of the `pbrt` rendering system, not just a high-level description of it.

Literate programs are written in a metalanguage that mixes a document formatting language (e.g., TEX or HTML) and a programming language (e.g., C++). Two separate systems process the program: a "weaver" that transforms the literate program into a document suitable for typesetting, and a "tangler" that produces source code suitable

for compilation. Our literate programming system is homegrown, but it was heavily influenced by Norman Ramsey's noweb system.

The literate programming metalanguage provides two important features. The first is the ability to mix prose with source code. This feature makes the description of the program just as important as its actual source code, encouraging careful design and documentation. Second, the language provides mechanisms for presenting the program code to the reader in an entirely different order than it is supplied to the compiler. Thus, the program can be described in a logical manner. Each named block of code is called a *fragment*, and each fragment can refer to other fragments by name.

As a simple example, consider a function `InitGlobals()` that is responsible for initializing all of a program's global variables:[1]

```
void InitGlobals(void) {
    num_marbles = 25.7;
    shoe_size = 13;
    dielectric = true;
    my_senator = REPUBLICAN;
}
```

Despite its brevity, this function is hard to understand without any context. Why, for example, can the variable `num_marbles` take on floating-point values? Just looking at the code, one would need to search through the entire program to see where each variable is declared and how it is used in order to understand its purpose and the meanings of its legal values. Although this structuring of the system is fine for a compiler, a human reader would much rather see the initialization code for each variable presented separately, near the code that actually declares and uses the variable.

In a literate program, one can instead write `InitGlobals()` like this:

⟨*Function Definitions*⟩ ≡
```
void InitGlobals() {
    ⟨Initialize Global Variables 2⟩
}
```

This defines a fragment, called ⟨*Function Definitions*⟩, that contains the definition of the `InitGlobals()` function. The `InitGlobals()` function itself refers to another fragment, ⟨*Initialize Global Variables*⟩. Because the initialization fragment has not yet been defined, we don't know anything about this function except that it will contain assignments to global variables. This is just the right level of abstraction for now, since no variables have been declared yet. When we introduce the global variable `shoe_size` somewhere later in the program, we can then write

⟨*Initialize Global Variables*⟩ ≡ 2
```
shoe_size = 13;
```

1 The example code in this section is merely illustrative and is not part of `pbrt` itself.

Here we have started to define the contents of ⟨*Initialize Global Variables*⟩. When the literate program is tangled into source code for compilation, the literate programming system will substitute the code `shoe_size = 13;` inside the definition of the `InitGlobals()` function. Later in the text, we may define another global variable, `dielectric`, and we can append its initialization to the fragment:

⟨*Initialize Global Variables*⟩ +≡ 2
```
dielectric = true;
```

The +≡ symbol after the fragment name shows that we have added to a previously defined fragment. When tangled, the result of these three fragments is the code

```
void InitGlobals() {
    shoe_size = 13;
    dielectric = true;
}
```

In this way, we can decompose complex functions into logically distinct parts, making them much easier to understand. For example, we can write a complicated function as a series of fragments:

⟨*Function Definitions*⟩ +≡
```
void complex_func(int x, int y, double *data) {
    ⟨Check validity of arguments⟩
    if (x < y) {
        ⟨Swap parameter values⟩
    }
    ⟨Do precomputation before loop⟩
    ⟨Loop through and update data array⟩
}
```

Again, the contents of each fragment are expanded inline in `complex_func()` for compilation. In the document, we can introduce each fragment and its implementation in turn. This decomposition lets us present code a few lines at a time, making it easier to understand. Another advantage of this style of programming is that by separating the function into logical fragments, each with a single and well-delineated purpose, each one can then be written and verified independently. In general, we will try to make each fragment less than 10 lines long.

In some sense, the literate programming system is just an enhanced macro substitution package tuned to the task of rearranging program source code. This may seem like a trivial change, but in fact literate programming is quite different from other ways of structuring software systems.

1.1.1 INDEXING AND CROSS-REFERENCING

The following features are designed to make the text easier to navigate. Indices in the page margins give page numbers where the functions, variables, and methods used on that page are defined. Indices at the end of the book collect all of these identifiers so that it's possible to find definitions by name. Appendix C, "Index of Fragments," lists the pages where each fragment is defined and the pages where it is used. Within the text, a defined

fragment name is followed by a list of page numbers on which that fragment is used. For example, a hypothetical fragment definition such as

⟨*A fascinating fragment*⟩ ≡ **184, 690**
```
num_marbles += .001;
```

indicates that this fragment is used on pages 184 and 690. If the fragments that use this fragment are not included in the book text, no page numbers will be listed. When a fragment is used inside another fragment, the page number on which it is first defined appears after the fragment name. For example,

⟨*Do something interesting*⟩+≡ **500**
```
InitializeSomethingInteresting();
⟨Do something else interesting 486⟩
CleanUp();
```

indicates that the ⟨*Do something else interesting*⟩ fragment is defined on page 486. If the definition of the fragment is not included in the book, no page number will be listed.

1.2 PHOTOREALISTIC RENDERING AND THE RAY-TRACING ALGORITHM

The goal of photorealistic rendering is to create an image of a 3D scene that is indistinguishable from a photograph of the same scene. Before we describe the rendering process, it is important to understand that in this context the word "indistinguishable" is imprecise because it involves a human observer, and different observers may perceive the same image differently. Although we will cover a few perceptual issues in this book, accounting for the precise characteristics of a given observer is a very difficult and largely unsolved problem. For the most part, we will be satisfied with an accurate simulation of the physics of light and its interaction with matter, relying on our understanding of display technology to present a good image to the viewer.

Most photorealistic rendering systems are based on the ray-tracing algorithm. Ray tracing is actually a very simple algorithm; it is based on following the path of a ray of light through a scene as it interacts with and bounces off objects in an environment. Although there are many ways to write a ray tracer, all such systems simulate at least the following objects and phenomena:

- *Cameras:* How and from where is the scene being viewed? Cameras generate rays from the viewing point into the scene.
- *Ray-object intersections:* We must be able to tell precisely where a given ray pierces a geometric object. In addition, we need to determine certain geometric properties of the object at the intersection point, such as a surface normal or its material. Most ray tracers also have some facility for finding the intersection of a ray with multiple objects, typically returning the closest intersection along the ray.
- *Light distribution:* Without lighting, there would be little point in rendering a scene. A ray tracer must model the distribution of light throughout the scene, including not only the locations of the lights themselves, but also the way in which they distribute their energy throughout space.

- *Visibility:* In order to know whether a given light deposits energy at a point on a surface, we must know whether there is an uninterrupted path from the point to the light source. Fortunately, this question is easy to answer in a ray tracer, since we can just construct the ray from the surface to the light, find the closest ray-object intersection, and compare the intersection distance to the light distance.
- *Surface scattering:* Each object must provide a description of its appearance, including information about how light interacts with the object's surface, as well as the nature of the reradiated (or *scattered*) light. We are usually interested in the properties of the light that is scattered directly toward the camera. Models for surface scattering are typically parameterized so that they can simulate a variety of appearances.
- *Recursive ray tracing:* Because light can arrive at a surface after bouncing off or passing through several other surfaces, it is usually necessary to trace additional rays originating at the surface to fully capture this effect. This is particularly important for shiny surfaces like metal or glass.
- *Ray propagation:* We need to know what happens to the light traveling along a ray as it passes through space. If we are rendering a scene in a vacuum, light energy remains constant along a ray. Although most human observers have never been in a vacuum, this is the typical assumption made by most ray tracers. More sophisticated models are available for tracing rays through fog, smoke, the Earth's atmosphere, and so on.

We will briefly discuss each of these simulation tasks in this section. In the next section, we will show `pbrt`'s high-level interface to the underlying simulation components and follow the progress of a single ray through the main rendering loop. We will also show one specific surface scattering model based on Turner Whitted's original ray-tracing algorithm.

1.2.1 CAMERAS

Nearly everyone has used a camera and is familiar with its basic functionality: you indicate your desire to record an image of the world (usually by pressing a button), and the image is recorded onto a piece of film or an electronic sensor. One of the simplest devices for taking photographs in the real world is called the *pinhole camera*. Pinhole cameras consist of a light-tight box with a tiny hole at one end (Figure 1.1). When the hole is uncovered, light enters this hole and falls on a piece of photographic paper that is affixed to the other end of the box. Despite its simplicity, this kind of camera is still used today, frequently for artistic purposes. Very long exposure times are necessary to get enough light on the film to form an image.

Although most cameras are substantially more complex than the pinhole camera, it is a convenient starting point for simulation. The most important function of the camera is to define the portion of the scene that will be recorded onto the film. In Figure 1.1, it is easy to see that connecting the pinhole to the edges of the film creates a double pyramid that extends into the scene. Objects that are not inside this pyramid cannot be imaged onto the film. Because modern cameras image a more complex shape than a pyramid, we will refer to the region of space that can potentially be imaged onto the film as the *viewing volume*.

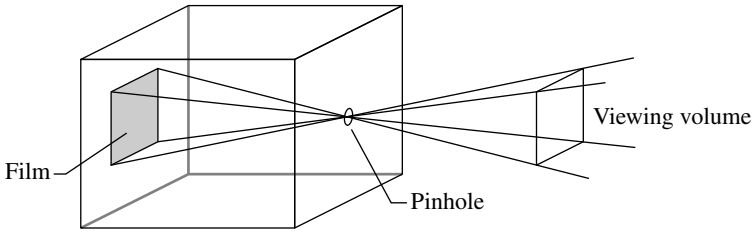


Figure 1.1: A Pinhole Camera.

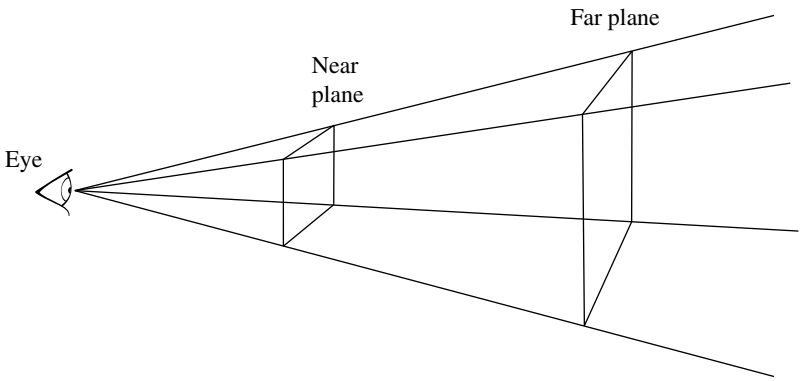


Figure 1.2: When we simulate a pinhole camera, we place the film in front of the hole at the near plane, and the hole is renamed the *eye*.

Another way to think about the pinhole camera is to place the film plane in *front* of the pinhole, but at the same distance (Figure 1.2). Note that connecting the hole to the film defines exactly the same viewing volume as before. Of course, this is not a practical way to build a real camera, but for simulation purposes it is a convenient abstraction. When the film (or image) plane is in front of the pinhole, the pinhole is frequently referred to as the *eye*.

Now we come to the crucial issue in rendering: at each point in the image, what color value do we display? If we recall the original pinhole camera, it is clear that only light rays that travel along the vector between the pinhole and a point on the film can contribute to that film location. In our simulated camera with the film plane in front of the eye, we are interested in the amount of light traveling from the image point to the eye.

Therefore, the task of the camera simulator is to take a point on the image and generate *rays* along which light is known to contribute to that image location. Because a ray consists of an origin point and a direction vector, this is particularly simple for the

pinhole camera model of Figure 1.2: it uses the pinhole for the origin, and the vector from the pinhole to the near plane as the ray's direction. For more complex camera models involving multiple lenses, the calculation of the ray that corresponds to a given point on the image may be more involved. However, if the process of converting image locations to rays is completely encapsulated in the camera module, the rest of the rendering system can focus on evaluating the lighting along those rays, and a variety of camera models can be supported. `pbrt`'s camera abstraction is described in detail in Chapter 6.

1.2.2 RAY-OBJECT INTERSECTIONS

Each time the camera generates a ray, the first task of the renderer is to determine which object, if any, that ray intersects first and where the intersection occurs. This intersection point is the visible point along the ray, and we will want to simulate the interaction of light with the object at this point. To find the intersection, we must test the ray for intersection against all objects in the scene and select the one that the ray intersects first. Given a ray r, we first start by writing it in *parametric form*:

$$\mathrm{r}(t) = \mathrm{o} + t\mathbf{d},$$

where o is the ray's origin, $\mathbf{d}$ is its direction vector, and t is a parameter whose legal range is $[0, \infty)$. We can obtain a point along the ray by specifying its parametric t value and evaluating the above equation.

It is often easy to find the intersection between the ray r and a surface defined by an implicit function $F(x, y, z) = 0$. We first substitute the ray equation into the implicit equation, producing a new function whose only parameter is t. We then solve this function for t and substitute the smallest positive root into the ray equation to find the desired point. For example, the implicit equation of a sphere centered at the origin with radius r is

$$x^2 + y^2 + z^2 - r^2 = 0,$$

so substituting the ray equation, we have

$$\left(\mathrm{o} + t\mathbf{d}\right)_x^2 + \left(\mathrm{o} + t\mathbf{d}\right)_y^2 + \left(\mathrm{o} + t\mathbf{d}\right)_z^2 - r^2 = 0.$$

This is just a quadratic equation in t, so we can easily solve it. If there are no real roots, the ray must miss the sphere. If there are roots, we select the smaller positive one to find the intersection point.

The intersection point is not enough information for the rest of the ray tracer; it needs to know certain properties of the surface at the point. First, the appearance model needs to be extracted and passed along to later stages of the ray-tracing algorithm, and additional geometric information about the intersection point will also be required in order to shade the point. For example, the surface normal $\mathbf{n}$ is always required. Although many ray tracers operate with only $\mathbf{n}$, more sophisticated rendering systems like `pbrt` require even more information, such as various partial derivatives of position and surface normal with respect to the local parameterization of the surface.

Of course, most scenes are made up of multiple objects. The brute-force intersection approach would be to test the ray against each object in turn, choosing the minimum t value of the intersections found. This approach, while correct, is very slow, even for

scenes of modest complexity. A solution is to incorporate an *acceleration structure* that quickly rejects whole groups of objects during the ray intersection process. This ability to quickly cull irrelevant geometry means that ray tracing frequently runs in $O(I \log N)$ time, where I is the number of pixels in the image and N is the number of objects in the scene.[2] (Building the acceleration structure is necessarily at least *O(N)* time.)

The geometric interface supported by `pbrt` is described in Chapter 3, and the acceleration interface is shown in Chapter 4.

1.2.3 LIGHT DISTRIBUTION

The ray-object intersection stage gives us a point to be shaded and some information about the geometry at that point. Recall that our eventual goal is to find the amount of light leaving this point in the direction of the camera. In order to do this, we need to know how much light is *arriving* at this point. This involves both the *geometric* and *radiometric* distribution of light in the scene. For very simple light sources (e.g., point lights), the geometric distribution of lighting is a simple matter of knowing the position of the lights. However, point lights do not exist in the real world, and so physically based lighting is often based on *area* light sources. This means that the light source is associated with a geometric object that emits illumination from its surface. However, we will use point lights in this section to illustrate the components of light distribution; rigorous discussion of light measurement and distribution is the topic of Chapters 5 and 12.

We frequently would like to know the amount of light energy being deposited on the differential area surrounding the intersection point (Figure 1.3). We will assume that the point light source has some power Φ associated with it, and that it radiates light equally in all directions. This means that the total amount of energy on a sphere surrounding the light is $\Phi/(4\pi)$. (These measurements will be explained and formalized in Chapter 12.) If we consider two such spheres (Figure 1.4), it is clear that the energy at a point on the larger sphere must be less than the energy at a point on the smaller sphere because the same total energy is distributed over a larger area. Specifically, the amount of energy arriving at a point on a sphere of radius r is proportional to $1/r^2$. Furthermore, it can be shown that if the tiny surface patch $\mathrm{d}A$ is tilted by an angle θ away from the vector from the surface point to the light, the amount of energy deposited on $\mathrm{d}A$ is proportional to $\cos\theta$. Putting this all together, the total light energy $\mathrm{d}E$ (the *differential irradiance*) deposited on $\mathrm{d}A$ is

$$\mathrm{d}E = \frac{\Phi \cos\theta}{4\pi r^2}.$$

Readers already familiar with basic lighting in computer graphics will notice two familiar laws encoded in this equation: the cosine falloff of light for tilted surfaces mentioned above, and the one-over-r-squared falloff of light with distance.

2 Although ray tracing's logarithmic complexity is often heralded as one of its key strengths, this is typically only true on average. The computational geometry literature has shown ray-tracing algorithms that have guaranteed logarithmic running time, but these algorithms only work for certain types of scenes and have very expensive preprocessing and storage requirements. Szirmay-Kalos and Márton provide pointers to the relevant literature (Szirmay-Kalos and Márton 1998). In practice, the ray intersection algorithms presented in this book are sublinear, but without expensive preprocessing and huge memory usage it is always possible to construct worst-case scenes where ray tracing runs in $O(IN)$ time.

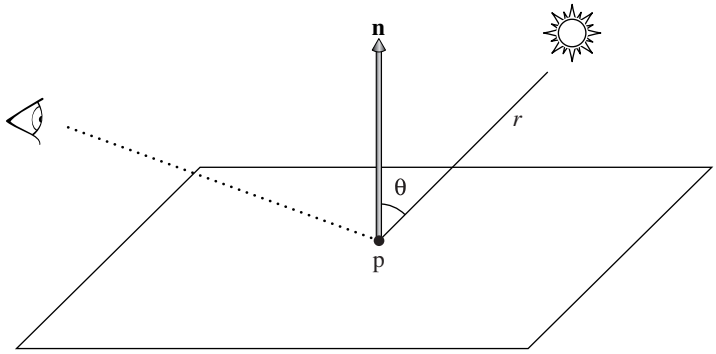


Figure 1.3: Geometric construction for evaluating the light energy at a point due to a point light source. The distance from the point to the light source is denoted by r.

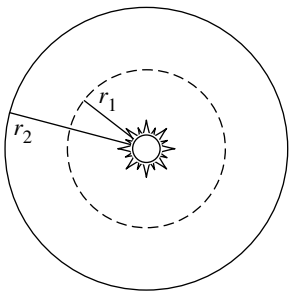


Figure 1.4: Since the point light radiates light equally in all directions, the same total energy is deposited on all spheres centered at the light.

Scenes with multiple lights are easily handled because illumination is *linear:* the contribution of each light can be computed separately and summed to obtain the overall contribution.

1.2.4 VISIBILITY

The lighting distribution described in the previous section ignores one very important component: *shadows*. Each light contributes illumination to the point being shaded only if the path from the point to the light's position is unobstructed (Figure 1.5).

Fortunately, in a ray tracer it is easy to determine if the light is visible from the point being shaded. We simply construct a new ray whose origin is at the surface point and whose direction points toward the light. These special rays are called *shadow rays*. If we trace this ray through the environment, we can check to see whether any intersections are found between the ray's origin and the light source by comparing the parametric t value of any

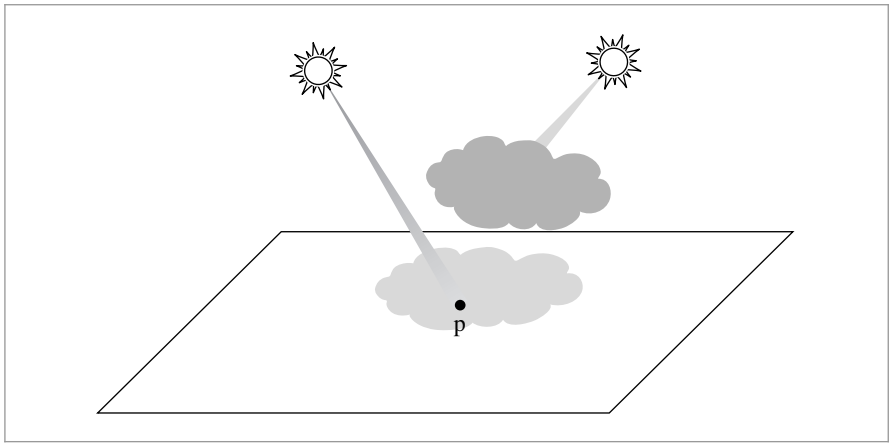


Figure 1.5: A light source only deposits energy on a surface if the source is not obscured as seen from the receiving point. The light source on the left illuminates the point p, but the light source on the right does not.

intersections found to the parameteric t value along the ray of the light source position. If there is no blocking object between the light and the surface, the light's contribution is included.

1.2.5 SURFACE SCATTERING

We now are able to compute two pieces of information that are vital for proper shading of a point: its location and the incident lighting.[3] Now we need to determine how the incident lighting is *scattered* at the surface. Specifically, we are interested in the amount of light energy scattered back along the ray that we originally traced to find the intersection point, since that ray leads to the camera (Figure 1.6).

Each object in the scene provides a *material*, which is a description of its appearance properties at each point on the surface. This description is given by the *Bidirectional Reflectance Distribution Function* (BRDF). This function tells us how much energy is reflected from a given incoming direction ω_i to a given outgoing direction ω_o. We will write the BRDF at p as $f_r(p, \omega_o, \omega_i)$. Now, computing the amount of light L scattered back toward the camera is straightforward:

```
for each light:
    if light is not blocked:
        incident_light = light.L(point)
        amount_reflected =
            surface.BRDF(hit_point, camera_vector, light_vector)
        L += amount_reflected * incident_light
```

3 Readers already familiar with rendering might object that the discussion in this section considers only direct lighting. Rest assured that `pbrt` does support global illumination.

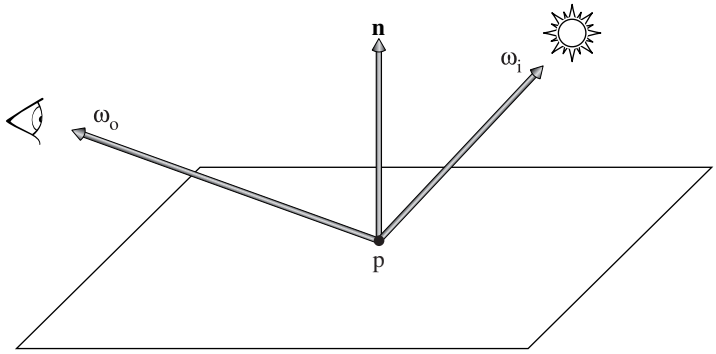


Figure 1.6: The Geometry of Surface Scattering. Incident light arriving along direction ω_i interacts with the surface at point p and is scattered back toward the camera along direction ω_o. The amount of light scattered toward the camera is given by the product of the incident light energy and the BRDF.

Here we are using L to represent the light; this represents a slightly different unit for light measurement than dE, which was used before.

It is easy to generalize the notion of a BRDF to transmitted light (obtaining a BTDF) or to general scattering of light arriving from either side of the surface. A function that describes general scattering is called a *Bidirectional Scattering Distribution Function* (BSDF). `pbrt` supports a variety of both physically and phenomenologically based BSDF models; they are described in Chapter 8.

1.2.6 RECURSIVE RAY TRACING

Turner Whitted's original paper on ray tracing emphasized its *recursive* nature. For example, if a ray from the camera hits a shiny object like a mirror, we can *reflect* the ray about the surface normal at the intersection point and recursively invoke the ray-tracing routine to find the light arriving at the point on the mirror, adding its contribution to the original camera ray. This same technique can be used to trace transmitted rays that intersect transparent objects. For a long time, most early ray-tracing examples showcased mirrors and glass balls (Figure 1.7) because these types of effects were difficult to capture with other rendering techniques.

In general, the amount of light that reaches the camera from a point on an object is given by the sum of light emitted by the object (if it is itself a light source) and the amount of reflected light. This idea is formalized by the *light transport equation* (also often known as the *rendering equation*), which says that the outgoing radiance $L_o(p, \omega_o)$ from a point p in direction ω_o is the emitted radiance at that point in that direction, $L_e(p, \omega_o)$, plus the incident radiance from all directions on the sphere $\mathcal{S}^2$ around p scaled by the BSDF $f(p, \omega_o, \omega_i)$ and a cosine term:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\mathcal{S}^2} f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos \theta_i|\, d\omega_i. \quad [1.1]$$

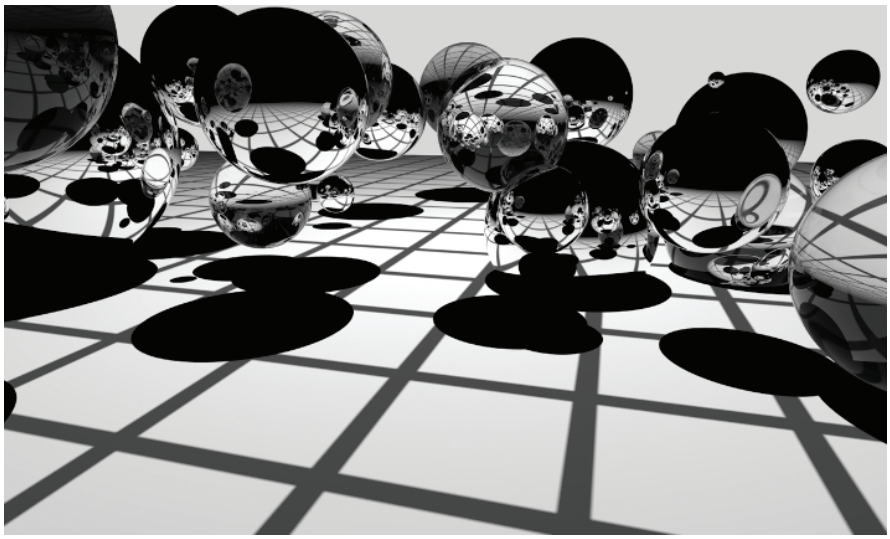


Figure 1.7: A Prototypical Example of Early Ray Tracing. Note the use of mirrored and glass objects, which emphasize the algorithm's ability to handle these kinds of surfaces.

We will show a more complete derivation of this equation in Sections 5.6.1 and 15.2. Solving this integral analytically is not possible except for the simplest of scenes, so we must either make simplifying assumptions or use numerical integration techniques.

Whitted's algorithm simplifies this integral by ignoring incoming light from most directions and only evaluating $L_i(p, \omega_i)$ for directions to light sources and for the directions of perfect reflection and refraction. In other words, it turns the integral into a sum over a small number of directions.

Whitted's method can be extended to capture more effects than just perfect mirrors and glass. For example, by tracing many recursive rays near the mirror-reflection direction and averaging their contributions, we obtain an approximation of glossy reflection. In fact, we can *always* recursively trace a ray whenever we hit an object. For example, we can randomly choose a reflection direction ω_i and weight the contribution of this newly spawned ray by evaluating the BRDF $f_r(p, \omega_o, \omega_i)$. This simple but powerful idea can lead to very realistic images because it captures all of the interreflection of light between objects. Of course, we need to know when to terminate the recursion, and choosing directions completely at random may make the rendering algorithm slow to converge to a reasonable result. These problems can be addressed, however; these issues are the topic of Chapters 13 to 15.

When we trace rays recursively in this manner, we are really associating a *tree* of rays with each image location (Figure 1.8), with the ray from the camera at the root of this tree. Note that each ray in this tree can have a *weight* associated with it; this allows us to model, for example, shiny surfaces that do not reflect 100% of the incoming light.

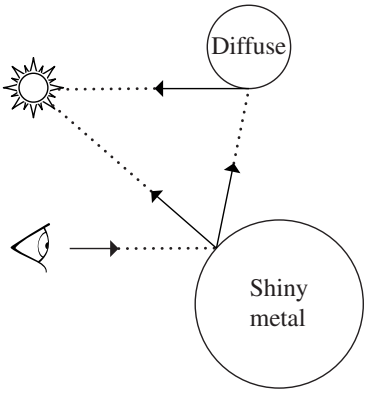


Figure 1.8: Recursive ray tracing associates an entire tree of rays with each image location.

1.2.7 RAY PROPAGATION

The prior discussion has assumed that rays are traveling through a vacuum. For example, when describing the distribution of light from a point source, we assumed that the energy was distributed equally on the surface of a sphere centered at the light without decreasing along the way. The presence of *participating media* such as smoke, fog, or dust can invalidate this assumption. Many ray tracers ignore these phenomena, though doing so is quite limiting. Even if we are not making a rendering of a smoke-filled room, almost all outdoor scenes are affected substantially by participating media. For example, Earth's atmosphere causes objects that are farther away to appear less saturated (Figure 1.9).

There are two ways in which a participating medium can affect the light propagating along a ray. First, the medium can *extinguish* (or *attenuate*) light, either by absorbing it or by scattering it in a different direction. We can capture this effect by computing the *transmittance* T between the ray origin and the intersection point. The transmittance tells us how much of the light scattered at the intersection point makes it back to the ray origin.

A participating medium can also add to the light along a ray. This can happen either if the medium emits light (as with a flame) or if the medium scatters light from other directions back along the ray (Figure 1.10). We can find this quantity by numerically evaluating the *volume light transport equation*, in the same way we evaluated the light transport equation to find the amount of light reflected from a surface. We will leave the description of participating media and volume rendering until Chapters 11 and 16. For now, it will suffice to say that we can compute the effect of participating media and incorporate its effect into the amount of light carried by the ray.

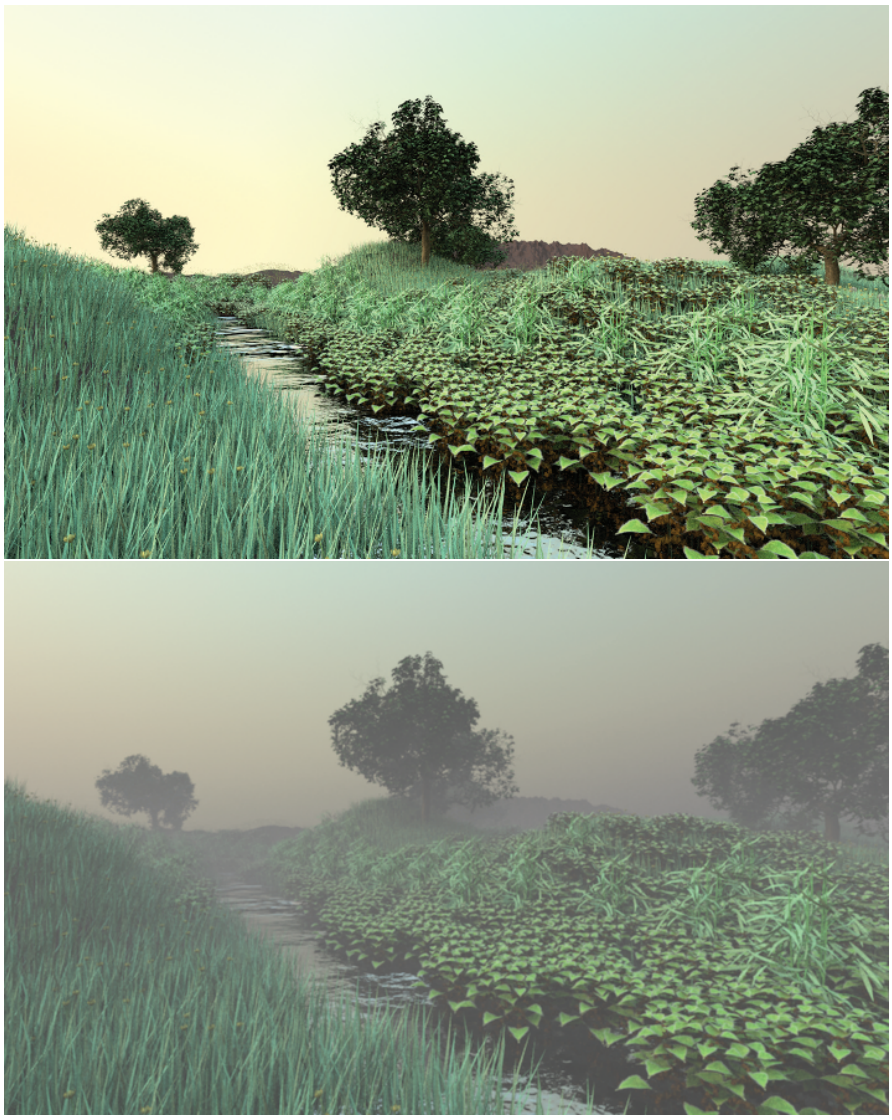


Figure 1.9: Earth's Atmosphere Decreases Saturation with Distance. The scene on the top is rendered without simulating this phenomenon, while the scene on the bottom includes an atmospheric model. This sort of atmospheric attenuation is an important depth cue when viewing real scenes and adds a sense of scale to the rendering on the bottom.

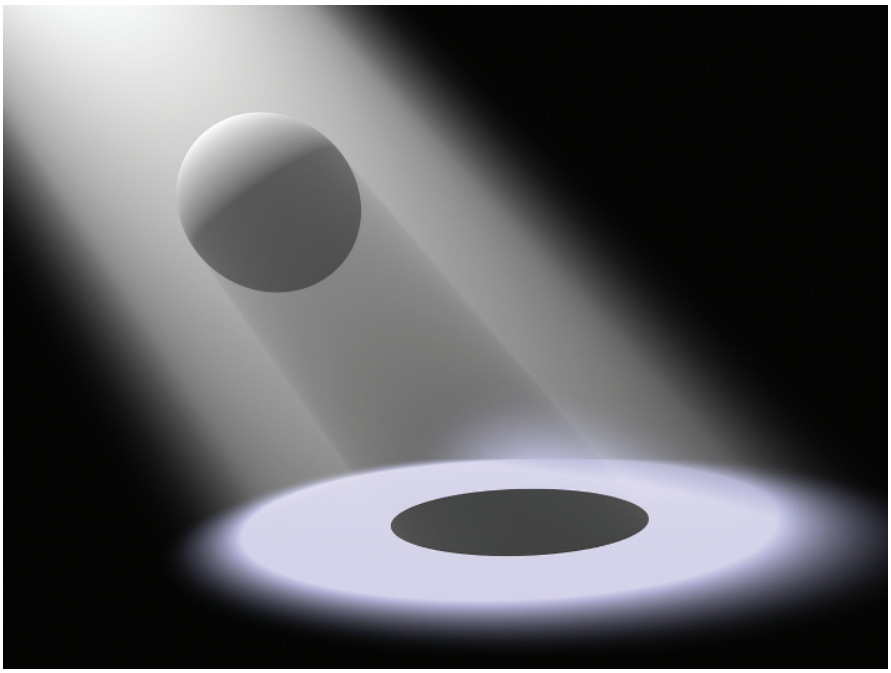


Figure 1.10: A Spotlight Shining on a Sphere through Fog. Notice that the shape of the spotlight's lighting distribution and the sphere's shadow are clearly visible due to the additional scattering in the participating medium.

1.3 pbrt: SYSTEM OVERVIEW

`pbrt` is structured using standard object-oriented techniques: abstract base classes are defined for important entities (for example, a `Shape` abstract base class defines the interface that all geometric shapes must implement, the `Light` abstract base class acts similarly for lights, etc.). The majority of the system is implemented purely in terms of the interfaces provided by these abstract base classes; for example, the code that checks for occluding objects between a light source and a point being shaded calls the `Shape` intersection methods and doesn't need to consider the particular types of shapes that are present in the scene. This approach makes it easy to extend the system, as adding a new shape only requires implementing a class that implements the `Shape` interface and linking it into the system.

`pbrt` is written using a total of 13 key abstract base classes, summarized in Table 1.1. Adding a new implementation of one of these types to the system is straightforward; the implementation must inherit from the appropriate base class, be compiled and linked into the executable, and the object creation routines in Appendix B must be modified to create instances of the object as needed as the scene description file is parsed. Section B.4 discusses extending the system in this manner in more detail.

Table 1.1: Main Interface Types. Most of pbrt is implemented in terms of 13 key abstract base classes, listed here. Implementations of each of these can easily be added to the system to extend its functionality.

Base class	Directory	Section
Shape	shapes/	3.1
Aggregate	accelerators/	4.2
Camera	cameras/	6.1
Sampler	samplers/	7.2
Filter	filters/	7.7
Film	film/	7.8
Material	materials/	9.2
Texture	textures/	10.3
VolumeRegion	volumes/	11.3
Light	lights/	12.1
Renderer	renderers/	1.3.3
SurfaceIntegrator	integrators/	Ch. 15 intro
VolumeIntegrator	integrators/	16.2

The source code to pbrt is distributed across a small directory hierarchy that can be found in the pbrt distribution, available from *pbrt.org/downloads.php*. All of the code for the pbrt core is in the src/core/ directory, and the main() function is contained in the short file src/main/pbrt.cpp. Various implementations of instances of the abstract base classes are in separate directories: src/shapes/ has implementations of the Shape base class, src/materials/ has implementations of Material, and so forth.

Throughout this section are a number of images rendered with extended versions of pbrt. Of them, Figures 1.11 through 1.14 are notable: not only are they visually impressive, but each of them was created by a student in a rendering course where the final class project was to extend pbrt with new functionality in order to render an interesting image. These images are among the best from those courses. Figures 1.15 and 1.16 were rendered with *LuxRender*, a GPL-licensed rendering system originally based on the pbrt source code from the first edition of the book. (See *www.luxrender.net* for more information about *LuxRender*.)

Aggregate 192
Camera 302
Film 403
Filter 393
Light 606
main() 20
Material 483
Renderer 24
Sampler 340
Shape 108
SurfaceIntegrator 740
Texture 519
VolumeIntegrator 876
VolumeRegion 587

1.3.1 PHASES OF EXECUTION

pbrt can be conceptually divided into two phases of execution. First, it parses the scene description file provided by the user. The scene description is a text file that specifies the geometric shapes that make up the scene, their material properties, the lights that illuminate them, where the virtual camera is positioned in the scene, and parameters to all of the individual algorithms used throughout the system. Each statement in the input file has a direct mapping to one of the routines in Appendix B; these routines comprise the procedural interface for describing a scene to pbrt. A number of example scenes are provided in the scenes/ directory in the pbrt distribution. The scene file format is



Figure 1.11: Guillaume Poncin and Pramod Sharma extended `pbrt` in numerous ways, implementing a number of complex rendering algorithms, to make this prize-winning image for Stanford's cs348b rendering competition. The trees are modeled procedurally with L-systems, a glow image processing filter increases the apparent realism of the lights on the tree, snow was modeled procedurally with metaballs, and a subsurface scattering algorithm gave the snow its realistic appearance by accounting for the effect of light that travels beneath the snow for some distance before leaving it.

specified in the file `docs/fileformat.pdf` and a user's guide to the file format is in the file `docs/usersguide.pdf`.

The end results of the parsing phase are an instance of the `Scene` class and an instance of the `Renderer` class. The `Scene` specifies the contents of the scene (geometric objects, lights, etc.), and the `Renderer` implements an algorithm to render it.

Once the scene has been specified, the second phase of execution begins and the main rendering loop executes. This phase is where `pbrt` usually spends the majority of its running time, and most of this book describes code that executes during this phase. The rendering loop is implemented in an implementation of the `Renderer::Render()` method, which is the focus of Section 1.3.4.

For the `SamplerRenderer` described in this chapter, the `Render()` method determines the light arriving at a virtual film plane for a large number of rays in order to model the process of image formation. After the contributions of all of these film samples have been computed, the final image is written to disk. The scene description data in memory are deallocated and the renderer then resumes processing statements from the scene description file until no more remain, allowing the user to specify another scene to be rendered, if desired.

`Renderer` 24
`Renderer::Render()` 24
`SamplerRenderer` 25
`Scene` 22



Figure 1.12: Abe Davis, David Jacobs, and Jongmin Baek rendered this amazing image of an ice cave to take the grand prize in the 2009 Stanford CS348b rendering competition. They first implemented a simulation of the physical process of glaciation, the process where snow falls, melts, and refreezes over the course of many years, forming stratified layers of ice. They then simulated erosion of the ice due to melted water runoff before generating a geometric model of the ice. Scattering of light inside the volume was simulated with volumetric photon mapping; the blue color of the ice is entirely due to modeling the wavelength-dependent absorption of light in the ice volume.

1.3.2 SCENE REPRESENTATION

`pbrt`'s `main()` function can be found in the file `main/pbrt.cpp`. This function is quite simple; it first loops over the provided command-line arguments in `argv`, initializing values in the `Options` structure and storing the filenames provided in the arguments. Running `pbrt` with `--help` as a command line argument prints all of the options that can be specified on the command line. The fragment that parses the command-line arguments, ⟨*Process command-line arguments*⟩, is straightforward and therefore not included in the book here.

`main()` 20

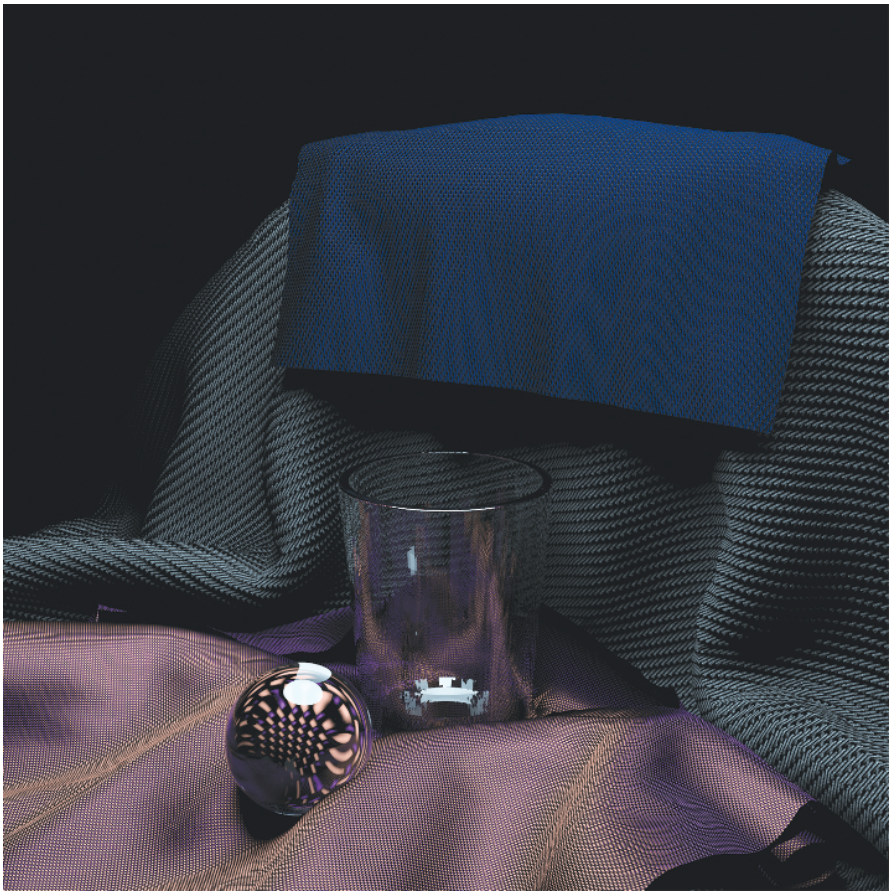


Figure 1.13: Lingfeng Yang implemented a bidirectional texture function to simulate the appearance of cloth, adding an analytic self-shadowing model, to render this image that took first prize in the 2009 Stanford CS348b rendering competition.

The options structure is then passed `pbrtInit()`, which does systemwide initialization. The `main()` function then parses the given scene description(s), leading to the creation of a `Scene` object that represents all of the elements (shapes, lights, etc.) that make up the scene and a `Renderer` object that implements an algorithm to render the scene. Because the input file can specify multiple scenes to be rendered, rendering actually begins as soon as the appropriate input directive is parsed. After all rendering is done, `pbrtCleanup()` does final cleanup before the system exits.

pbrtCleanup() 1052
pbrtInit() 1051
Renderer 24
Scene 22

The `pbrtInit()` and `pbrtCleanup()` functions appear in a *mini-index* in the page margin, along with the number of the page where they are actually defined. The mini-indices have pointers to the definitions of almost all of the functions, classes, methods, and member variables used or referred to on each page.



Figure 1.14: Jared Jacobs and Michael Turitzin added an implementation of Kajiya and Kay's texel-based fur rendering algorithm (Kajiya and Kay 1989) to `pbrt` and rendered this image, where both the fur on the dog and the shag carpet are rendered with the texel fur algorithm.

⟨*main program*⟩ ≡

```
int main(int argc, char *argv[]) {
    Options options;
    vector<string> filenames;
    ⟨Process command-line arguments⟩
    pbrtInit(options);
    ⟨Process scene description 21⟩
    pbrtCleanup();
    return 0;
}
```

Options 1051
pbrtCleanup() 1052
pbrtInit() 1051

If `pbrt` is run with no input filenames provided, then the scene description is read from standard input. Otherwise it loops through the provided filenames, processing each file in turn.



Figure 1.15: This contemporary indoor scene was modeled and rendered by Florent Boyer (*www.florentboyer.com*). The image was rendered using *LuxRender*, a GPL licensed physically-based rendering system originally based on `pbrt`'s source code. Modelling and texturing was done using Blender.

⟨*Process scene description*⟩ ≡ 20
```
if (filenames.size() == 0) {
    ⟨Parse scene from standard input 21⟩
} else {
    ⟨Parse scene from input files 21⟩
}
```

The `ParseFile()` function parses a scene description file, either from standard input or from a file on disk; it returns `false` if it was unable to open the file. The mechanics of parsing scene description files will not be described in this book; the parser implementation can be found in the `lex` and `yacc` files `core/pbrtlex.ll` and `core/pbrtparse.yy`, respectively. Readers who want to understand the parsing subsystem but are not familiar with these tools may wish to consult Levine, Mason, and Brown (1992). We use the common UNIX idiom that a file named "`-`" represents standard input:

⟨*Parse scene from standard input*⟩ ≡ 21
```
ParseFile("-");
```

If a particular input file can't be opened, the `Error()` routine reports this information to the user. `Error()` uses the same format string semantics as `printf()`.

⟨*Parse scene from input files*⟩ ≡ 21
```
for (u_int i = 0; i < filenames.size(); i++)
    if (!ParseFile(filenames[i]))
        Error("Couldn't open scene file \"%s\"", filenames[i].c_str());
```

`Error()` 1005
`ParseFile()` 21



Figure 1.16: Martin Lubich modeled this scene of the Austrian Imperial Crown and rendered it using *LuxRender*, an open-source fork of the `pbrt` codebase. The scene was modeled in Blender and consists of approximately 1.8 million vertices. It is illuminated by six area light sources with emission spectra based on measured data from a real-world light source and was rendered with 1280 samples per pixel in 73 hours of computation on a quad-core CPU. See Martin's Web site, *www.loramel.net*, for more information, including downloadable Blender scene files.

As the scene file is parsed, objects are created that represent the lights and geometric primitives in the scene. These are all stored in the `Scene` object, which is created by the `RenderOptions::MakeScene()` method in Section B.3.7 in Appendix B. The `Scene` class is declared in `core/scene.h` and defined in `core/scene.cpp`.

We will not include the implementation of the `Scene` constructor here; it just stores copies of its arguments in the various member variables inside the class.

⟨*Scene Declarations*⟩ ≡

```
class Scene {
public:
    ⟨Scene Public Methods 23⟩
    ⟨Scene Public Data 23⟩
};
```

Each geometric object in the scene is represented by a `Primitive`, which combines two objects: a `Shape` that specifies its geometry, and a `Material` that describes its appearance (e.g., the object's color, whether it has a dull or glossy finish). All of these geometric

Material 483
Primitive 185
RenderOptions::
MakeScene() 1072
Scene 22
Shape 108

primitives are collected into a single aggregate `Primitive` in the `Scene` member variable `Scene::aggregate`. This aggregate is a special kind of primitive that itself holds references to many other primitives. Because it implements the `Primitive` interface it appears no different than a single primitive to the rest of the system. The specific class used to implement `Scene::aggregate` stores all the scene's primitives in an acceleration data structure that reduces the number of unnecessary ray intersection tests with primitives that a given ray doesn't pass near.

⟨*Scene Public Data*⟩ ≡ 22
```
Primitive *aggregate;
```

Each light source in the scene is represented by a `Light` object, which specifies the shape of a light and the distribution of energy that it emits. The `Scene` stores all of the lights in a `vector` class from the C++ standard library. While some renderers support separate light lists per geometric object, allowing a light to illuminate only some of the objects in the scene, this idea does not map well to the physically based rendering approach taken in `pbrt`, so we use only this per-scene list.

⟨*Scene Public Data*⟩ +≡ 22
```
vector<Light *> lights;
```

In addition to geometric primitives, `pbrt` also supports participating media, or *volumetric* primitives. These types of primitives are supported through the `VolumeRegion` interface. The system's support for participating media is described in Chapter 11. Like `Primitives`, multiple `VolumeRegions` are all stored together in a single aggregate region, `Scene::volumeRegion`.

⟨*Scene Public Data*⟩ +≡ 22
```
VolumeRegion *volumeRegion;
```

The `Scene` class provides a handful of additional methods. Its `Intersect()` method traces the given ray into the scene and returns a Boolean value indicating whether the ray intersected any of the primitives. If so, it fills in the provided `Intersection` structure with information about the closest intersection point along the ray. The `Intersection` structure is defined in Section 4.1.

Intersection 186
Light 606
Primitive 185
Primitive::Intersect() 186
Ray 66
Scene 22
Scene::aggregate 23
Scene::Intersect() 23
Scene::IntersectP() 24
Scene::volumeRegion 23
VolumeRegion 587

⟨*Scene Public Methods*⟩ ≡ 22
```
bool Intersect(const Ray &ray, Intersection *isect) const {
    bool hit = aggregate->Intersect(ray, isect);
    return hit;
}
```

A closely related method is `Scene::IntersectP()`, which checks for the existence of intersections along the ray, but does not return any information about those intersections. Because this routine doesn't need to search for the closest intersection or compute any additional information about the intersections, it is generally more efficient than `Scene::Intersect()`. This routine is used for shadow rays.

⟨*Scene Public Methods*⟩ +≡ 22
```
bool IntersectP(const Ray &ray) const {
    bool hit = aggregate->IntersectP(ray);
    return hit;
}
```

Finally, `Scene::WorldBound()` returns a 3D box that bounds all of the geometry in the scene, which is simply the bounding box of `Scene::aggregate`. The `Scene` class caches this bound to avoid having to repeatedly compute it.

⟨*Scene Constructor Implementation*⟩ ≡
```
bound = aggregate->WorldBound();
if (volumeRegion) bound = Union(bound, volumeRegion->WorldBound());
```

⟨*Scene Public Data*⟩ +≡ 22
```
BBox bound;
```

⟨*Scene Method Definitions*⟩ ≡
```
const BBox &Scene::WorldBound() const {
    return bound;
}
```

1.3.3 RENDERER INTERFACE AND `SamplerRenderer`

Rendering an image of the scene is handled by an instance of a class that implements the `Renderer` interface. `Renderer` is an abstract base class that defines a few methods that must be provided by all renderers. It is defined in the files `core/renderer.h` and `core/renderer.cpp`. In this section, we will define both the interface that all `Renderers` must provide as well as the start of one instance of a `Renderer`, the `SamplerRenderer`.

⟨*Renderer Declarations*⟩ ≡
```
class Renderer {
public:
    ⟨Renderer Interface 24⟩
};
```

The main method that `Renderers` must provide is `Render()`; the `Renderer` is passed a pointer to a `Scene` and computes an image of the scene or more generally, a set of measurements of the scene lighting. For example, in Section 17.3, we define a completely different kind of `Renderer` that computes measurements of incident illumination at a set of points in the scene and writes the results to a text file, without generating an image at all. These measurements can then be used for interactive rendering of the scene, among other applications.

⟨*Renderer Interface*⟩ ≡ 24
```
virtual void Render(const Scene *scene) = 0;
```

`Renderers` are also required to provide methods that compute information about the illumination along rays in the scene. `Li()` returns the incident radiance along the given ray. In addition to the ray and the scene, it takes a number of additional parameters: the `Sample` (which may be `NULL`) provides random sample values for Monte Carlo integration

`BBox` 70
`Primitive::IntersectP()` 186
`Primitive::WorldBound()` 185
`Ray` 66
`Renderer` 24
`Sample` 343
`SamplerRenderer` 25
`Scene` 22
`Scene::aggregate` 23
`Scene::bound` 24
`Scene::volumeRegion` 23
`Scene::WorldBound()` 24
`Union()` 72

computations in the integrator, and the RNG is a pseudo-random number generator that is also available for this purpose. The MemoryArena performs efficient allocation of small temporary amounts of memory that may be needed while computing radiance along the ray. Finally, information about the ray's geometric intersection point can be returned via the Intersection, if its pointer is non-NULL, and the volumetric transmittance along the ray is returned via the T parameter, also if non-NULL.

⟨*Renderer Interface*⟩ +≡ 24
```
virtual Spectrum Li(const Scene *scene, const RayDifferential &ray,
    const Sample *sample, RNG &rng, MemoryArena &arena,
    Intersection *isect = NULL, Spectrum *T = NULL) const = 0;
```

Transmittance() returns the fraction of light that is attenuated by volumetric scattering along the ray. Renderer implementations will generally dispatch to Integrators (defined in Chapters 15 and 16) to compute the values returned by Li() and Transmittance().

⟨*Renderer Interface*⟩ +≡ 24
```
virtual Spectrum Transmittance(const Scene *scene,
    const RayDifferential &ray, const Sample *sample,
    RNG &rng, MemoryArena &arena) const = 0;
```

Now we will define the SamplerRenderer implementation of the Renderer interface and show how its Render() method computes an image of the scene. SamplerRenderer is so named because its rendering process is driven by a stream of *samples* from a Sampler; each such sample identifies a point on the image at which to compute the arriving light to form the image. The definition of SamplerRenderer is in the files renderers/samplerrenderer.h and renderers/samplerrenderer.cpp.

⟨*SamplerRenderer Declarations*⟩ ≡
```
class SamplerRenderer : public Renderer {
public:
    ⟨SamplerRenderer Public Methods⟩
private:
    ⟨SamplerRenderer Private Data 25⟩
};
```

Camera 302
Film 403
Integrator 740
Intersection 186
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Sampler 340
SamplerRenderer 25
Scene 22
Spectrum 263

The SamplerRenderer stores a pointer to a Sampler. The role of this class is subtle, but its implementation can substantially affect the quality of the images that the system generates. First, the sampler is responsible for choosing the points on the image plane from which rays are traced. Second, it is responsible for supplying the sample positions used by the integrators in their light transport computations; for example, some integrators need to choose random points on light sources to compute illumination from area lights. Generating a good distribution of these samples is an important part of the rendering process that can substantially affect overall efficiency and is discussed in Chapter 7.

⟨*SamplerRenderer Private Data*⟩ ≡ 25
```
Sampler *sampler;
```

The Camera object controls the viewing and lens parameters such as position, orientation, focus, and field of view. A Film member variable inside the Camera class handles image storage. The Camera classes are described in Chapter 6, and Film is described in

Section 7.8. The Film is responsible for writing the final image to disk and possibly displaying it on the screen as it is being computed.

⟨*SamplerRenderer Private Data*⟩ +≡ 25
```
Camera *camera;
```

Integrators handle the task of simulating the propagation of light in the scene in order to compute how much light arrives at image sample positions on the film plane. They are so named because they numerically evaluate the integrals in the surface and volume light transport equations that describe the distribution of light in the environment. SurfaceIntegrators compute reflected light from geometric surfaces, while VolumeIntegrators handle the scattering from volumetric primitives. Integrators are described in Chapters 15 and 16.

⟨*SamplerRenderer Private Data*⟩ +≡ 25
```
SurfaceIntegrator *surfaceIntegrator;
VolumeIntegrator *volumeIntegrator;
```

The SamplerRenderer constructor just stores pointers to these objects in member variables. The SamplerRenderer is created in the RenderOptions::MakeRenderer() method, which is in turn called by pbrtWorldEnd(), which is called by the input file parser when it is done parsing a scene description from an input file and is ready to render the scene.

⟨*SamplerRenderer Method Definitions*⟩ ≡
```
SamplerRenderer::SamplerRenderer(Sampler *s, Camera *c,
        SurfaceIntegrator *si, VolumeIntegrator *vi) {
    sampler = s;
    camera = c;
    surfaceIntegrator = si;
    volumeIntegrator = vi;
}
```

Camera 302
Film 403
pbrtWorldEnd() 1071
Renderer 24
Renderer::Render() 24
RenderOptions::MakeRenderer() 1072
Sampler 340
SamplerRenderer 25
SamplerRenderer::camera 26
SamplerRenderer::sampler 25
SamplerRenderer::surfaceIntegrator 26
SamplerRenderer::volumeIntegrator 26
Scene 22
SurfaceIntegrator 740
VolumeIntegrator 876

1.3.4 THE MAIN RENDERING LOOP

After the Scene and the Renderer have been allocated and initialized, the Renderer::Render() method is invoked, starting the second phase of pbrt's execution: the main rendering loop. In the SamplerRenderer's implementation of this method, at each of a series of positions on the image plane, the method uses the Camera and the Sampler to generate a ray into the scene, and then uses its integrators to determine the amount of light arriving at the image plane along that ray. This value is passed to the Film, which records the light's contribution. Figure 1.17 summarizes the main classes used in this method and the flow of data among them.

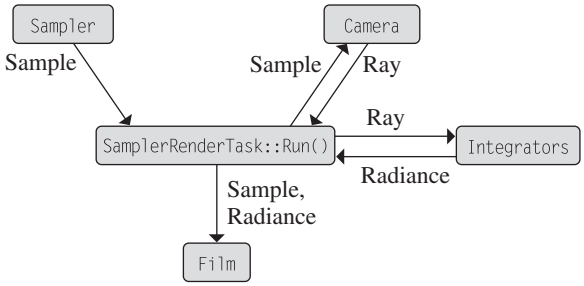


Figure 1.17: Class Relationships for the Main Rendering Loop in the `SamplerRenderer::Render()` Method in `renderers/sample.cpp`. The `Sampler` provides a sequence of sample values, one for each image sample to be taken. The `Camera` turns a sample into a corresponding ray from the film plane, and the `Integrators` compute the radiance along that ray arriving at the film. The sample and its radiance are given to the `Film`, which stores their contribution in an image. This process repeats until the `Sampler` has provided as many samples as are necessary to generate the final image.

⟨*SamplerRenderer Method Definitions*⟩ +≡
```
void SamplerRenderer::Render(const Scene *scene) {
    ⟨Allow integrators to do preprocessing for the scene 27⟩
    ⟨Allocate and initialize sample 27⟩
    ⟨Create and launch SamplerRendererTasks for rendering image 28⟩
    ⟨Clean up after rendering and store final image 29⟩
}
```

Before rendering can begin, the `SamplerRenderer` calls the `Preprocess()` methods of the integrators. Because information like the specific light sources and geometry of the scene aren't available when the integrators are first created, the `Preprocess()` method gives them an opportunity to do any necessary scene-dependent initialization or preprocessing. For example, the `PhotonIntegrator` in Section 15.6 uses this opportunity to create data structures that hold a representation of the distribution of illumination in the scene.

⟨*Allow integrators to do preprocessing for the scene*⟩ ≡ 27
```
surfaceIntegrator->Preprocess(scene, camera, this);
volumeIntegrator->Preprocess(scene, camera, this);
```

Before rendering starts, the `Render()` method also creates a `Sample` object, into which the `Sampler` will store the samples it generates during the main loop. Because the number and types of samples that need to be generated are partially dependent on the integrators, the `Sample` constructor takes pointers to the integrators so they can be queried for their requirements. See Section 7.2.1 for more information about how integrators request particular sets of samples.

⟨*Allocate and initialize* `sample`⟩ ≡ 27
```
Sample *sample = new Sample(sampler, surfaceIntegrator,
                            volumeIntegrator, scene);
```

Camera 302
Film 403
Integrator 740
Integrator::Preprocess() 740
PhotonIntegrator 802
Sample 343
Sampler 340
SamplerRenderer 25
SamplerRenderer::Render() 27
Scene 22

Now the main rendering loop begins. So that rendering can proceed in parallel on systems with multiple processing cores, the process of generating the image is decomposed into a set of SamplerRendererTasks where each SamplerRendererTask is responsible for computing the samples in a small rectangular subset of the image. The SamplerRendererTask inherits from the Task abstract base class defined in Appendix A.9.4.

Tasks represent independent work items that are all immediately ready for execution; they can be run in any order, on any available processor. This decomposition is the foundation of how pbrt renders images in parallel. Section 1.3.5 discusses implications to the system's design due to this decomposition, and Appendix A.9 goes into more depth on performance issues related to parallelism and presents the classes and utility methods pbrt uses in its parallel implementation.

The EnqueueTasks() function takes an array of tasks and runs them on all of the processors in the system. It returns immediately to the calling thread, allowing the tasks to execute asynchronously; here there is no other work to do while the image is rendered, so the Render() method here immediately calls WaitForAllTasks(), which suspends the calling thread of execution until all of the enqueued tasks have finished.

⟨*Create and launch* SamplerRendererTasks *for rendering image*⟩ ≡ 27
```
⟨Compute number of SamplerRendererTasks to create for rendering 29⟩
vector<Task *> renderTasks;
for (int i = 0; i < nTasks; ++i)
    renderTasks.push_back(new SamplerRendererTask(scene, this, camera,
                                sampler, sample, nTasks-1-i, nTasks));
EnqueueTasks(renderTasks);
WaitForAllTasks();
for (uint32_t i = 0; i < renderTasks.size(); ++i)
    delete renderTasks[i];
```

There are two factors to trade off in deciding how many SamplerRendererTasks to create: load-balancing and per-task overhead. On one hand, we'd like to have significantly more tasks than there are processors in the system: consider a four-core computer running only four SamplerRendererTasks. In general, it's likely that some of the SamplerRendererTasks will take less processing time than others; the ones that are responsible for parts of the image where the scene is relatively simple will usually take less processing time than parts of the image where the scene is relatively complex. Therefore, if the number of tasks was equal to the number of processors, some processors would finish before others and sit idle while waiting for the processor that had the longest running task. Figure 1.18 illustrates this idea; it shows the distribution of execution time for the SamplerRendererTasks created to render the shiny sphere scene in Figure 1.7. The longest-running task took 151 times longer than the slowest.

On the other hand, having too many tasks is also inefficient. There is a (small) fixed overhead for launching each task, and the more tasks there are, the more times this overhead must be paid. The computation of nTasks here attempts to trade off these two factors. It ensures that there are at least 32 tasks for each processing core in the system while also computing how many tasks would be needed so that each one would work

EnqueueTasks() 1041
SamplerRendererTask 29
Task 1041
WaitForAllTasks() 1041

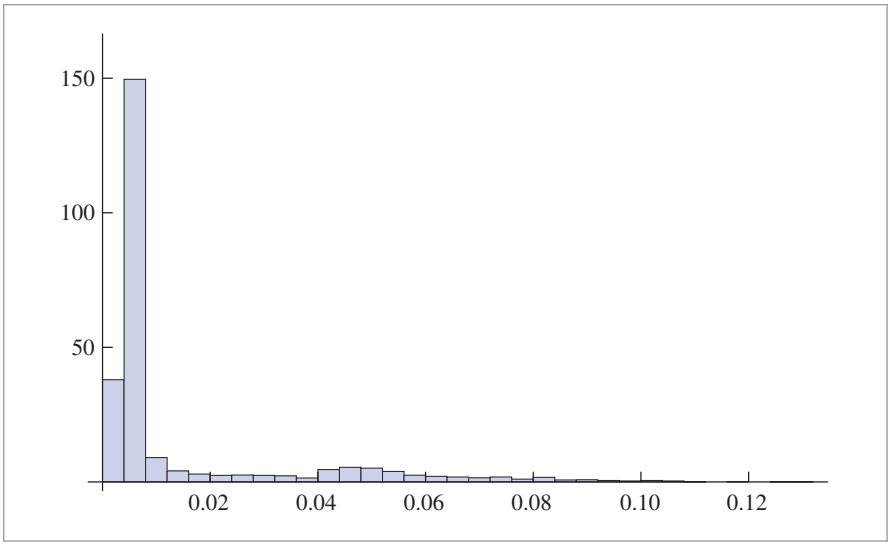


Figure 1.18: Histogram of Time Spent Running Each `SamplerRendererTask` for the Scene in Figure 1.7. The horizontal axis measures time in seconds. Note the wide variation in execution time for task execution, illustrating that different parts of the image required substantially different amounts of computation.

on an approximately 16 × 16-pixel block of the image. Whichever value gives a larger number of tasks is used. Finally, this value is rounded up to be a power of two, which will make it easier for the image decomposition code in the following to break the image into rectangular regions.

⟨*Compute number of* `SamplerRendererTask`*s to create for rendering*⟩ ≡ 28
```
int nPixels = camera->film->xResolution * camera->film->yResolution;
int nTasks = max(32 * NumSystemCores(), nPixels / (16*16));
nTasks = RoundUpPow2(nTasks);
```

After all of the rendering tasks have finished, the `SamplerRenderer`'s `Render()` method deletes the `Sample` and calls the `Film`'s method to write the image out to disk.

⟨*Clean up after rendering and store final image*⟩ ≡ 27
```
delete sample;
camera->film->WriteImage();
```

Now we will discuss the implementation of `SamplerRendererTask`.

⟨*SamplerRendererTask Declarations*⟩ ≡
```
class SamplerRendererTask : public Task {
public:
    ⟨SamplerRendererTask Public Methods 30⟩
private:
    ⟨SamplerRendererTask Private Data 30⟩
};
```

Camera::film 302
Film 403
Film::WriteImage() 404
Film::xResolution 403
Film::yResolution 403
NumSystemCores() 1041
RoundUpPow2() 1002
Sample 343
SamplerRenderer 25
SamplerRendererTask 29
Task 1041

Because the SamplerRendererTask runs independently of the SamplerRenderer::Render() method, all of the variables and data that it needs must be passed to its constructor. These values are all stored in SamplerRendererTask member variables for access when the task actually runs. This is a typical idiom for tasks in the system; we will usually elide this part of Task implementations for brevity. Note that it is also given a task number, taskNum, and the total number of tasks launched, taskCount. From these two values, the task will later be able to determine which part of the image it is responsible for computing.

⟨*SamplerRendererTask Public Methods*⟩ ≡ 29
```
SamplerRendererTask(const Scene *sc, Renderer *ren, Camera *c,
                    Sampler *ms, Sample *sam, int tn, int tc)
{
    scene = sc; renderer = ren; camera = c; mainSampler = ms;
    origSample = sam; taskNum = tn; taskCount = tc;
}
```

⟨*SamplerRendererTask Private Data*⟩ ≡ 29
```
const Scene *scene;
const Renderer *renderer;
Camera *camera;
Sampler *mainSampler;
Sample *origSample;
int taskNum, taskCount;
```

When the task system in Appendix A.9.4 decides to run this task on a particular processor, the SamplerRendererTask::Run() method will be called and the task can start its work. The task does a little bit of setup work, determining which part of the film plane it is responsible for and allocating space for some temporary data before using the Sampler to generate image samples, the Camera to determine corresponding rays leaving the film plane, and the Integrators to compute radiance along those rays arriving at the film.

⟨*SamplerRendererTask Definitions*⟩ ≡
```
void SamplerRendererTask::Run() {
    ⟨Get sub-Sampler for SamplerRendererTask 31⟩
    ⟨Declare local variables used for rendering loop 31⟩
    ⟨Allocate space for samples and intersections 31⟩
    ⟨Get samples from Sampler and update image 32⟩
    ⟨Clean up after SamplerRendererTask is done with its image region⟩
}
```

As the scene description is parsed, a Sampler that generates image and integration samples for the entire image is created. Here, the SamplerRendererTask uses the Sampler::GetSubSampler() method to get a new Sampler that only generates samples for the subset of the image that the SamplerRenderer is responsible for. The GetSubSampler() method uses the task number and the total number of tasks passed to the SamplerRendererTask constructor to determine which part of the image the returned subsampler should generate samples for. This method may return NULL, because work is decomposed by image pixels—for example, given a 1×1-pixel image, only one task will have any work to do. Any additional tasks that were created will just return immediately. (It would perhaps be

Camera 302
Integrator 740
Renderer 24
Sample 343
Sampler 340
SamplerRenderer::Render() 27
SamplerRendererTask 29
SamplerRendererTask::Run() 30
Scene 22
Task 1041

better to not create the extra tasks at all in this case, though this is an infrequent enough situation that it's just as well to catch it here.)

⟨*Get sub*-`Sampler` *for* `SamplerRendererTask`⟩ ≡ 30
```
Sampler *sampler = mainSampler->GetSubSampler(taskNum, taskCount);
if (!sampler)
    return;
```

During rendering, the `Run()` method needs two additional task-local variables: a `MemoryArena` and a `RNG`. The `MemoryArena` is passed to `Integrators` for allocating small amounts of temporary memory during rendering. Rather than call the system's regular memory allocation routines (e.g. `malloc()` or `new`) to do this, the system instead uses the `MemoryArena` class to manage pools of memory for this purpose. Using the `MemoryArena` for this purpose both enables higher-performance allocation than is provided by the standard library routines and also simplifies freeing allocated data. The `RNG` is made available to `Integrators` for generating pseudo-random numbers for Monte Carlo sampling.

Neither of these classes is safe to be used by multiple threads concurrently without additional synchronization; for efficiency, it's better to have unique instances of them for each task so that they can just be used directly.

⟨*Declare local variables used for rendering loop*⟩ ≡ 30
```
MemoryArena arena;
RNG rng(taskNum);
```

The `Sampler` may provide one or more sample values when asked to generate samples. For some sample-generation algorithms, it's easier to generate a set of sample values all at once than to generate them one at a time. The `Sampler::MaximumSampleCount()` method returns an upper bound on the number of samples it will return at once. Given this bound, that number of `Samples` are created to store the values returned by the `Sampler`. For each sample, it is necessary to store the corresponding ray (represented here by the `RayDifferential` class), spectra for the radiance along the ray `Ls`, the transmittance along the ray `Ts`, and an `Intersection` object for each ray intersection. All of these arrays are deallocated at the end of the `Render()` method.

⟨*Allocate space for samples and intersections*⟩ ≡ 30
```
int maxSamples = sampler->MaximumSampleCount();
Sample *samples = origSample->Duplicate(maxSamples);
RayDifferential *rays = new RayDifferential[maxSamples];
Spectrum *Ls = new Spectrum[maxSamples];
Spectrum *Ts = new Spectrum[maxSamples];
Intersection *isects = new Intersection[maxSamples];
```

Each time through the loop, `Sampler::GetMoreSamples()` is called to initialize the `samples` array with one or more image sample values; this method returns the number of samples it initialized, or zero when it has finished generating all of the samples for the region of the image that it is responsible for. The fragments in the loop body find the corresponding camera ray for each sample and pass it to the integrators to compute the radiance along the ray arriving at the film plane. Finally, they add the samples' contributions to the final image and free any temporary memory allocated in the `MemoryArena` during the course

Integrator 740
Intersection 186
MemoryArena 1015
RayDifferential 69
RNG 1003
Sample 343
Sample::Duplicate() 346
Sampler 340
Sampler::
GetMoreSamples() 340
Sampler::GetSubSampler() 341
Sampler::
MaximumSampleCount() 341
SamplerRendererTask::
mainSampler 30
SamplerRendererTask::
origSample 30
SamplerRendererTask::
taskCount 30
SamplerRendererTask::
taskNum 30
Spectrum 263

of computing results for the rays. (The MemoryArena here therefore shouldn't be used to allocate any memory with a longer lifetime than needed to compute the result for a single ray.)

⟨*Get samples from* Sampler *and update image*⟩ ≡ 30
```
int sampleCount;
while ((sampleCount = sampler->GetMoreSamples(samples, rng)) > 0) {
    ⟨Generate camera rays and compute radiance along rays 32⟩
    ⟨Report sample results to Sampler, add contributions to image 33⟩
    ⟨Free MemoryArena memory from computing image sample values 33⟩
}
```

⟨*Generate camera rays and compute radiance along rays*⟩ ≡ 32
```
for (int i = 0; i < sampleCount; ++i) {
    ⟨Find camera ray for sample[i] 32⟩
    ⟨Evaluate radiance along camera ray 33⟩
}
```

The Camera interface provides two main methods: Camera::GenerateRay(), which returns the ray for a given image sample position, and Camera::GenerateRayDifferential(), which returns a *ray differential*, which incorporates information about the rays that the Camera would generate for samples that are one pixel away on the image plane in both the x and y directions. Ray differentials are used to get better results from some of the texture functions defined in Chapter 10, making it possible to compute how quickly a texture varies with respect to the pixel spacing, a key component of texture antialiasing. While the Ray class holds just the origin and direction of a single ray, RayDifferential inherits from Ray so that it has not only those member variables but also two additional Rays, rx and ry, to hold these neighbors. One important detail is that the direction vector of the generated ray must be of unit length; most of the integrators depend on this property.

The GenerateRayDifferential() method is passed a Sample and a pointer to a ray differential; it initializes all of the fields of the RayDifferential based on the contents of the sample to give the differential rays as if a single sample is being taken for each pixel (i.e., with an implicit assumption that image samples are spaced one pixel apart). However, when rendering high-quality images, many samples are often averaged together to compute each pixel value. Therefore, the ScaleDifferentials() methods scales the differential rays to account for the actual spacing between samples on the film plane.

The camera also returns a floating-point weight associated with the ray. For simple camera models, each ray is weighted equally, but more complex Cameras that more accurately model the process of image formation by lens systems may generate some rays that contribute more than others. Such a camera model might simulate the effect of less light arriving at the edges of the film plane than at the center, an effect called *vignetting*. The returned weight will be used here to scale the ray's contribution to the image.

⟨*Find camera ray for* sample[i]⟩ ≡ 32
```
float rayWeight = camera->GenerateRayDifferential(samples[i], &rays[i]);
rays[i].ScaleDifferentials(1.f / sqrtf(sampler->samplesPerPixel));
```

Camera 302
Camera::GenerateRay() 303
Camera::GenerateRayDifferential() 303
MemoryArena 1015
Ray 66
RayDifferential 69
RayDifferential::ScaleDifferentials() 70
Sampler 340
Sampler::GetMoreSamples() 340
Sampler::samplesPerPixel 340
SamplerRendererTask::camera 30

Now that we have a ray, the next task is to determine the amount of light arriving at the image plane along that ray (its *radiance*). The usual notation for radiance arriving along a ray is L_i, so the method to compute radiance is `Renderer::Li()`. Radiance values are represented here with the `Spectrum` class, which is `pbrt`'s abstraction for energy distributions that vary over wavelength—in other words, *color*.

⟨*Evaluate radiance along camera ray*⟩ ≡ 32
```
if (rayWeight > 0.f)
    Ls[i] = rayWeight * renderer->Li(scene, rays[i], &samples[i], rng,
                                     arena, &isects[i], &Ts[i]);
else {
    Ls[i] = 0.f;
    Ts[i] = 1.f;
}
```
⟨*Issue warning if unexpected radiance value returned*⟩

A common side effect of bugs in the rendering process is that impossible radiance values are computed. For example, division by zero results in radiance values equal either to the IEEE floating-point infinity or "not a number" value. The renderer looks for this possibility, as well as for spectra with negative contributions, and prints an error message when it encounters them. Here we won't include the fragment that does this, ⟨*Issue warning if unexpected radiance value returned*⟩. See the implementation in `renderers/samplerrenderer.cpp` if you're interested in its details.

After the radiance carried by the rays is known, the image can be updated. Before this happens, the `Sampler::ReportResults()` method is used to pass the radiance values and information about the intersections found back to the `Sampler`; this gives the sampler a chance to incorporate information from the results of these samples into the samples it generates later. (For example, it could generate extra samples in pixels that have a lot of detail.) This method returns `true` if this group of samples should be added to the image, or `false` if it should be discarded. For some adaptive sampling algorithms, the sampler may want to discard the initial set of samples and generate new ones in their stead.

If the samples are to be added to the image, the `Film::AddSample()` method updates the pixels in the image given the results from a sample. The details of this process are explained in Sections 7.7 and 7.8.

Camera::film 302
Film::AddSample() 403
MemoryArena 1015
MemoryArena::FreeAll() 1017
Renderer::Li() 25
Sampler 340
Sampler::ReportResults() 341
SamplerRendererTask::camera 30
SamplerRendererTask::renderer 30
SamplerRendererTask::scene 30
Spectrum 263

⟨*Report sample results to* `Sampler`*, add contributions to image*⟩ ≡ 32
```
if (sampler->ReportResults(samples, rays, Ls, isects, sampleCount))
    for (int i = 0; i < sampleCount; ++i)
        camera->film->AddSample(samples[i], Ls[i]);
```

After processing a group of samples, all of the allocated memory in the `MemoryArena` is freed together when `MemoryArena::FreeAll()` is called. Section 9.1.1 discusses one use of the `MemoryArena` during rendering, and Section A.5.4 has the `MemoryArena` implementation.

⟨*Free* `MemoryArena` *memory from computing image sample values*⟩ ≡ 32
```
arena.FreeAll();
```

We won't include the final code fragment from SamplerRendererTask::Run(), ⟨*Clean up after* SamplerRendererTask *is done with its image region*⟩ here, as it just frees the memory allocated for samples, rays, and intersections.

Recall that the SamplerRenderer::Li() method is called by the SamplerRendererTask::Run() method to compute the radiance along the given ray. It is given a Sample to pass along to the integrators for their use in their computations and a MemoryArena for their temporary memory needs. Information about the intersection point and the transmittance due to volume scattering along the ray may be returned to the caller, if non-NULL values are passed for the corresponding parameters.

The Scene::Intersect() method finds the first surface that the ray intersects by passing the request on to an accelerator Primitive. The accelerator performs ray-primitive intersection tests with the geometric Primitives that the ray potentially intersects, using each shape's Shape::Intersect() routine. If an intersection is found, this method returns true and returns a filled-in Intersection object. The SamplerRenderer::Li() method then calls SurfaceIntegrator::Li() to compute the outgoing radiance L_o from the first surface that the ray intersects and then stores the result in Li. It then invokes VolumeIntegrator::Transmittance() to compute the fraction of light T that is extinguished between the surface and the camera due to participating media. Finally, VolumeIntegrator::Li() determines the radiance L_v added along the ray due to interactions with participating media. The net effect of these interactions is $TL_i + L_v$; this calculation is explained further in Section 16.1.

⟨*SamplerRenderer Method Definitions*⟩ +≡

```
Spectrum SamplerRenderer::Li(const Scene *scene,
        const RayDifferential &ray, const Sample *sample, RNG &rng,
        MemoryArena &arena, Intersection *isect, Spectrum *T) const {
    ⟨Allocate local variables for isect and T if needed 35⟩
    Spectrum Li = 0.f;
    if (scene->Intersect(ray, isect))
        Li = surfaceIntegrator->Li(scene, this, ray, *isect, sample,
                                   rng, arena);
    else {
        ⟨Handle ray that doesn't intersect any geometry 35⟩
    }
    Spectrum Lvi = volumeIntegrator->Li(scene, this, ray, sample, rng,
                                        T, arena);
    return *T * Li + Lvi;
}
```

If the caller does pass NULL values for isect and T parameters, then scratch instances of these objects are allocated on the stack in this method so that the integrators can just assume that the corresponding values are non-NULL.

Intersection 186
MemoryArena 1015
Primitive 185
RayDifferential 69
RNG 1003
Sample 343
SamplerRenderer 25
SamplerRenderer::Li() 34
SamplerRenderer::surfaceIntegrator 26
SamplerRenderer::volumeIntegrator 26
SamplerRendererTask 29
SamplerRendererTask::Run() 30
Scene 22
Scene::Intersect() 23
Shape::Intersect() 111
Spectrum 263
SurfaceIntegrator::Li() 741
VolumeIntegrator::Li() 876
VolumeIntegrator::Transmittance() 876

⟨*Allocate local variables for* `isect` *and* `T` *if needed*⟩ ≡ 34
```
Spectrum localT;
if (!T) T = &localT;
Intersection localIsect;
if (!isect) isect = &localIsect;
```

Rays that don't hit any geometry may still carry radiance—certain types of light sources may not be associated with any geometry but can still contribute radiance to rays that do not hit anything. For example, the Earth's sky illuminates points on the Earth's surface with blue light, even though there isn't geometry associated with the sky per se. Therefore, for rays that do not hit anything, each light's `Light::Le()` method is called so that these particular lights can contribute to the ray's radiance. Most light sources will not contribute in this way, but in certain cases this is very useful. See Section 12.5 for an example of such a light.

⟨*Handle ray that doesn't intersect any geometry*⟩ ≡ 34
```
for (uint32_t i = 0; i < scene->lights.size(); ++i)
   Li += scene->lights[i]->Le(ray);
```

It is also useful to be able to independently compute the attenuation along a ray due to participating media; the `SamplerRenderer::Transmittance()` method computes this quantity by forwarding the request to the `VolumeIntegrator::Transmittance()` method.

⟨*SamplerRenderer Method Definitions*⟩ +≡
```
Spectrum SamplerRenderer::Transmittance(const Scene *scene,
        const RayDifferential &ray, const Sample *sample, RNG &rng,
        MemoryArena &arena) const {
    return volumeIntegrator->Transmittance(scene, this, ray, sample,
                                           rng, arena);
}
```

1.3.5 PARALLELIZATION OF `pbrt`

As of the writing of the second edition of this book, it's difficult to buy a new laptop or desktop computer with only one processing core. The computer systems of today and of the future will increasingly feature multiple processing cores, both on CPUs and on highly parallel throughput processors like GPUs. This development in computer architecture places a new burden on software developers, as significant increases in performance over time are now only available to programs that can run in parallel, with many separate threads of execution performing computation simultaneously.

This burden is a substantial one; parallel programming is notoriously difficult. When two computations that the programmer believes are independent are executing simultaneously but then interact unexpectedly, the program may crash or generate unexpected results. However, the bug may not manifest itself again if the program is run again, perhaps due to those particular computations not running simultaneously during the next

Intersection 186
Light::Le() 631
MemoryArena 1015
RayDifferential 69
RNG 1003
Sample 343
SamplerRenderer 25
SamplerRenderer::
Transmittance() 35
SamplerRenderer::
volumeIntegrator 26
Scene 22
Scene::lights 23
Spectrum 263
VolumeIntegrator::
Transmittance() 876

run. Fortunately, increasingly good tools to help developers find these sorts of interactions are being developed.[4]

For a parallel program to scale well to increasingly large numbers of processors, it needs to be able to provide a substantial amount of independent computation to the system. However, any computation dependent on results of prior computation can't be run concurrently with the computation it depends on. Some researchers have claimed that ray tracing is in the class of *embarrassingly parallel* applications. This observation is partially correct, in that there is in fact an enormous amount of independent computation available in most ray-tracing algorithms; each pixel could potentially be rendered by a separate processor, offering scaling up to computers with millions of processors. However, as we'll see at a number of places in the implementation of `pbrt`, the parallelization of a reasonably complete software system is more difficult than the parallelization of a small proof-of-concept research system.

We assume that the system has some number of processing *cores*, each of which can run a separate thread of execution. A *task system*, implemented in Appendix A.9.4, launches a hardware thread on each core in the system. These threads poll a *task queue* for work. As soon as code in `pbrt` adds tasks to the task queue, processing cores can start to remove tasks from the queue and execute them.

In `pbrt` we assume that the computation is running on processors that provide *coherent shared memory*. (This is true of today's CPUs and is likely to continue to be provided by future CPUs.) The main idea is that all of the threads can read and write to a common set of memory locations and that changes to memory made by one thread will be seen by other threads. This property greatly simplifies the implementation of the system as there's no need to explicitly communicate data between tasks.

Data Races and Coordination

Although coherent shared memory relieves tasks of the need to explicitly communicate data with each other, they still need to *coordinate* their access to shared data; a danger of coherent shared memory is *data races*. If two threads of execution modify the same location in memory without coordination between the two of them, the program will almost certainly compute incorrect results or even crash. Consider the example of two processors simultaneously running the following innocuous-looking code, where `globalCounter` starts with a value of two:

```
extern int globalCounter;
if (--globalCounter == 0) {
    printf("done!\n");
}
```

Because the two threads don't coordinate their reading and writing of `globalCounter`, it is possible that "done" will be printed zero, one, or even two times! The assembly instructions generated by the compiler are likely to correspond to steps something like the following:

4 We found the open source tool `helgrind`, part of the `valgrind` suite of tools, instrumental for helping to find bugs in the `pbrt` implementation as we were developing it.

```
extern int globalCounter;
int temp = globalCounter;
temp = temp - 1;
globalCounter = temp;
if (globalCounter == 0) {
    printf("done!\n");
}
```

Now, consider different ways this code could be executed on two processors. For example, the second processor could start executing slightly after the first one, but the first one could go idle for a few cycles after executing the first few instructions:

```
Thread A                        Thread B
int temp = globalCounter;
temp = temp - 1;                (idle)
globalCounter = temp;
                                int temp = globalCounter;
(idle)                          temp = temp - 1;
                                globalCounter = temp;
if (globalCounter == 0) {       if (globalCounter == 0) {
    printf("done!\n");             printf("done!\n");
}                               }
```

Many unpredictable events can cause these sorts of execution bubbles, ranging from the OS interrupting the thread to cache misses.

In this ordering, both threads read the value of zero from `globalCounter` and both execute the `printf()` call. In this case, the error is not fatal, but if instead the system was freeing resources in the `if` block, then it would attempt to free the same resources twice, which would very likely cause a crash. Consider now this potential execution order:

```
Thread A                        Thread B
int temp = globalCounter;       int temp = globalCounter;
temp = temp - 1;                temp = temp - 1;
globalCounter = temp;           (idle)
(idle)                          globalCounter = temp;
if (globalCounter == 0) {       if (globalCounter == 0) {
    printf("done!\n");             printf("done!\n");
}                               }
```

In this case, `globalCounter` ends up with a value of one, and neither thread executes the `if` block. These examples illustrate the principle that when multiple threads of execution are accessing shared modified data, they must somehow synchronize their access. Two main mechanisms are available for doing this type of synchronization: mutual exclusion and atomic operations.

Mutex 1038

MutexLock 1039

Mutual exclusion is implemented with `Mutex` objects in `pbrt`. A `Mutex` can be used to protect access to a memory location, ensuring that only one thread can update it at a time. Consider the following updated version of the previous computation; here a `MutexLock`

object acquires a lock on the mutex and releases it when it goes out of scope at the final brace.

```
extern int globalCounter;
extern Mutex globalCounterMutex;
{ MutexLock lock(globalCounterMutex);
  int temp = globalCounter;
  temp = temp - 1;
  globalCounter = temp;
  if (globalCounter == 0) {
      printf("done!\n");
  }
}
```

If two threads are executing this code and try to acquire the mutex at the same time, then the mutex will allow only one of them to proceed, stalling the other one in the `MutexLock` constructor. Only when the first thread has finished the computation and its `MutexLock` goes out of scope, releasing the lock on the mutex, is the second thread able to acquire the mutex itself and continue the computation.

```
Thread A                                  Thread B
{ MutexLock lock(globalCounterMutex);    { MutexLock lock(globalCounterMutex);
  int temp = globalCounter;                  (stalled by mutex)
  .
  .
  .
} (mutex released)
                                             (mutex acquired)
                                             int temp = globalCounter;
                                             .
                                             .
                                             .
                                           } (mutex released)
```

With correct mutual exclusion here, the `printf()` will only be executed once, no matter what the ordering of execution between the two threads is.

Atomic memory operations (or *atomics*) are the other option for correctly performing this type of memory update with multiple threads. Atomics are a feature provided by almost all CPU architectures; they are machine instructions that guarantee that their respective memory updates will be performed in a single transaction. ("Atomic" in this case refers to the notion that the memory updates are indivisible.) The implementations of atomic operations in `pbrt` are defined in Appendix A.9.2. Using atomics, the computation above could be written to use the `AtomicAdd()` increment operation (incrementing by −1 in this case) as below:

```
extern AtomicInt32 globalCounter;
if (AtomicAdd(&globalCounter, -1) == 0) {
  printf("done!\n");
}
```

AtomicAdd() 1038
MutexLock 1039

`AtomicAdd()` adds the given value (here, -1) to the given variable (`globalCounter`) and returns the new value of the variable. Using an atomic operation ensures that if two threads simultaneously try to update the variable then not only will the final value of the variable be the expected value but each thread will be returned the value of the variable after its update alone. In this example, then, `globalCounter` will end up with a value of zero, as expected, with one thread guaranteed to have the value one returned from the `AtomicAdd()` call and the other thread guaranteed to have zero returned.

When there is a choice between atomic operations and mutexes, atomics are generally preferable, in that they are generally more efficient. However, if more than a single memory location needs to be updated then a mutex must generally be used.[5] Note that both of these approaches—atomics and mutexes—are conventions that the programmer must ensure are consistently adhered to. If just a single code-path updates a shared variable without consistently holding a mutex or using an atomic (whichever approach was chosen for updates to that location), then the program may fail under some circumstances.

Section A.9 in Appendix A has more information about parallel programming, with additional details on performance issues and pitfalls, as well as the various utility classes and routines used in the parallel implementation of `pbrt`.

Conventions in pbrt

In Section 1.3.4, we decomposed the process of rendering an image into a number of tasks, where each task was repsonsible for a small rectangular portion of the image (this approach is sometimes called *screen-space decomposition*). When rendering begins, each processor takes a task from the task queue and does the work associated with it, until no more tasks remain. These tasks are, by design, independent, so they can be executed in any order on however many processors are available in the system. However, care is necessary to ensure that there are no synchronization errors in the data accesses made by the tasks.

In `pbrt` (as is the case for most ray tracers) the vast majority of data at render-time is read only (for example, the scene description and texture maps). All of the parsing of the scene file and creation of the scene representation in memory is done with a single thread of execution, so there are no synchronization issues during that phase of execution. During rendering, concurrent read access to all of the read-only data by multiple threads works directly; we only need to be concerned with situations where data in memory is being updated.

Updating the output image is an important instance of a case where multiple tasks need to update shared data. Although the rendering work is distributed to tasks by image decomposition, under some circumstances tasks will need to update pixel data that is nominally being computed by other tasks—see Section 7.7 for the filtering theory behind this issue and Section 7.8.2 for the implementation of the image update code that deals with synchronization between threads. There, if we can be sure that only a single task will be updating the image data for a given pixel then the code just updates it directly, but if multiple tasks need to update it then atomic operations are used instead.

AtomicAdd() 1038

5 A generalization of atomic operations, *transactional memory*, makes possible more extensive sequences of memory updates, though it isn't yet available on mainstream processors.

Another important case of multiple tasks updating shared data is preprocessing work by some Integrators that build data structures for use during rendering before rendering begins. Some of them do substantial amounts of work during this phase, and in turn they break their preprocessing down into many tasks. See the implementations of the Preprocess() methods in Sections 15.5 (Irradiance Caching) and 15.6 (Photon Mapping) for examples.

The final instance of the shared data update issue is in data structures that are built on demand during rendering. For example, the GridAccel defined in Section 4.3 lazily builds a data structure for accelerating ray-primitive intersection computations. Because this data structure is shared by all of the currently executing tasks, updates to it during rendering need to be coordinated. (See in particular the use of a mutex in the grid traversal code in Section 4.3.2.)

When adding new code to pbrt, it's important to make sure to not inadvertently add code that modifies shared data without proper synchronization. In many cases, this can easily happen inadvertently—for example, when adding a new Shape implementation, the Shape will normally only perform read accesses to its member variables after it has been created. Sometimes, however, shared data may be inadvertently introduced. Consider the following code idiom, often seen in single-threaded code:

```
static bool firstCall = true;
if (firstCall) {
    .
    .  additional initialization
    .
    firstCall = false;
}
```

This code is unsafe with multiple threads of execution, as multiple threads may see the value of firstCall as true and all execute the initialization code. Writing this safely requires either atomic operations or mutexes. (This particular idiom probably needs a mutex, as it is likely that we would want other threads of execution that reach this test to stall until the additional intialization had been completed.)

Reentrancy Expectations in pbrt

Many methods of classes in pbrt are required to be *reentrant*—safe for multiple threads of execution to call concurrently. Particular instances of these methods must either be safe naturally due to not updating shared global data or using mutexes or atomic operations to safely perform any updates that are needed.

As a general rule, the low-level classes and structures in the system are not reentrant. For example, the Point class, which stores three float values to represent a point in 3D space, is not safe for multiple threads to call methods that modify it at the same time. (Multiple threads can use Points as read-only data simultaneously, of course.) The runtime overhead to make Point thread-safe would have a substantial effect on performance with little benefit in return.

The same is true for classes like Vector, Normal, Spectrum, Transform, Quaternion, and DifferentialGeometry. These classes are usually either created at scene construction time

DifferentialGeometry 102
GridAccel 196
Integrator 740
Normal 65
Point 63
Quaternion 92
Shape 108
Spectrum 263
Transform 76
Vector 57

and then used as read-only data or allocated on the stack during rendering and used only by a single thread.

The utility classes `MemoryArena` (used for high-performance temporary memory allocation) and `RNG` (pseudo-random number generation) are also not safe for use by multiple threads; these classes store state that is modified by their method calls, and the overhead from protecting modification to their state with mutual exclusion would be excessive relative to the amount of computation they perform. Consequently, in code like the `SamplerRendererTask::Run()` method above, the implementation allocates these as private per-thread instances of these classes allocated on the stack.

With two exceptions, implementations of the higher-level abstract base classes listed in Table 1.1 are all expected to be safe for multiple threads to use simultaneously. With a little care, it is usually straightforward to implement specific instances of these base classes so they don't modify any shared state in their methods.

The first exception is the `Integrator Preprocess()` and `RequestSamples()` methods. These are called by the system during scene construction, and implementations of them generally modify shared state in the `Integrator` implementation—for example, by building data structures that represent the distribution of illumination in the scene. Therefore, it's helpful to allow the implementor to assume that only a single thread will call into these methods. (This is a separate issue from the consideration that for implementations of these methods that are computationally intensive the implementation of them will often launch multiple `Task`s to do the computation in parallel.)

The second exception is the `Sampler::GetMoreSamples()` and `Sampler::ReportResults()` methods—these are also not expected to be thread safe. This is another instance where this requirement would impose an excessive performance and scalability impact; many threads simultaneously trying to get samples from a single `Sampler` would limit the system's overall performance. Therefore, as described in Section 1.3.4, a unique `Sampler` is created for each rendering task with `Sampler::GetSubSampler()`; this sampler can then be used by just the single task, without any mutual exclusion overhead.

All stand-alone functions in `pbrt` are reentrant (as long as multiple threads don't pass pointers to the same data to them).

1.3.6 AN INTEGRATOR FOR WHITTED RAY TRACING

Chapters 15 and 16 include the implementations of many different surface and volume integrators, based on a variety of algorithms with differing levels of accuracy. Here we will present a surface integrator based on Whitted's ray-tracing algorithm. This integrator accurately computes reflected and transmitted light from specular surfaces like glass, mirrors, and water, although it doesn't account for other types of indirect lighting effects like light bouncing off a wall and illuminating a room. The more complex integrators later in the book build on the ideas in this integrator to implement more sophisticated light transport algorithms. The `WhittedIntegrator` class can be found in the `integrators/whitted.h` and `integrators/whitted.cpp` files in the `pbrt` distribution.

Integrator 740
MemoryArena 1015
RNG 1003
Sampler 340
Sampler::
GetMoreSamples() 340
Sampler::GetSubSampler() 341
Sampler::ReportResults() 341
SamplerRendererTask::Run() 30
Task 1041
WhittedIntegrator 42

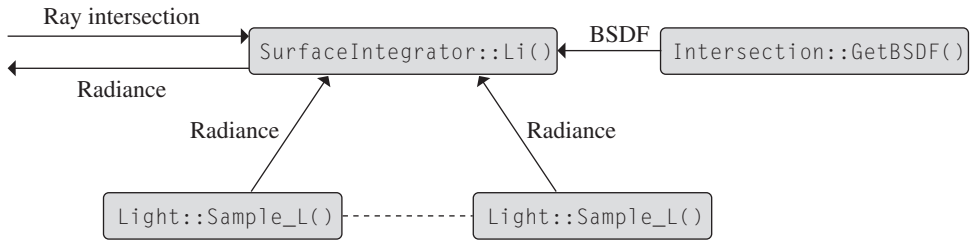


Figure 1.19: Class Relationships for Surface Integration. The main rendering loop passes a camera ray and information about its intersection point to the `SurfaceIntegrator`, which returns the radiance along that ray arriving at the film plane. The integrator finds the material properties at the intersection point in the form of a `BSDF` and uses the `Lights` in the `Scene` to determine the illumination there. Together, these give the information needed to compute the radiance reflected back along the ray at the intersection point.

⟨*WhittedIntegrator Declarations*⟩ ≡
```
class WhittedIntegrator : public SurfaceIntegrator {
public:
    ⟨WhittedIntegrator Public Methods⟩
private:
    ⟨WhittedIntegrator Private Data 42⟩
};
```

The key method that all surface integrators must provide is `SurfaceIntegrator::Li()`, which returns the radiance along a ray. Figure 1.19 summarizes the data flow among the main classes used during integration at surfaces.

⟨*WhittedIntegrator Method Definitions*⟩ ≡
```
Spectrum WhittedIntegrator::Li(const Scene *scene,
        const Renderer *renderer, const RayDifferential &ray,
        const Intersection &isect, const Sample *sample, RNG &rng,
        MemoryArena &arena) const {
    Spectrum L(0.);
    ⟨Compute emitted and reflected light at ray intersection point 43⟩
    return L;
}
```

The Whitted integrator works by recursively evaluating radiance along reflected and refracted ray directions. It stops the recursion at a predetermined maximum depth, `WhittedIntegrator::maxDepth`. By default, the maximum recursion depth is five. Without this termination criterion, the recursion might never terminate (imagine, for example, a hall-of-mirrors scene). This member variable is initialized based on parameters set in the scene description file in the `WhittedIntegrator` constructor, which we will not show here.

⟨*WhittedIntegrator Private Data*⟩ ≡ 42
```
int maxDepth;
```

BSDF 478
Intersection 186
Light 606
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Scene 22
Spectrum 263
SurfaceIntegrator 740
SurfaceIntegrator::Li() 741
WhittedIntegrator 42
WhittedIntegrator::
 maxDepth 42

⟨*Compute emitted and reflected light at ray intersection point*⟩ ≡ 42
```
⟨Evaluate BSDF at hit point 43⟩
⟨Initialize common variables for Whitted integrator 43⟩
⟨Compute emitted light if ray hit an area light source 43⟩
⟨Add contribution of each light source 45⟩
if (ray.depth + 1 < maxDepth) {
    ⟨Trace rays for specular reflection and refraction 46⟩
}
```

Recall that bidirectional scattering distribution functions describe how a surface reflects light arriving at its boundary; they essentially encapsulate the difference between the appearance of a mirror versus colored paint, and so forth. They are represented in pbrt by the BSDF class. pbrt provides BSDF implementations for several standard scattering functions used in computer graphics, such as Lambertian reflection and the Torrance–Sparrow microfacet model. These and other reflection models are described in Chapter 8.

The BSDF interface makes it possible to compute reflected light at a single surface point, but BSDFs may vary across a surface. Surfaces with complex material properties, such as wood or marble, have a different BSDF at each point. Even if wood is modeled as being perfectly diffuse, the color at each point will depend on the wood's grain. These spatial variations of shading parameters are described with Textures, which in turn may be described procedurally or stored in image maps (Chapter 10).

To obtain the BSDF at the hit point, the integrator calls the Intersection::GetBSDF() method:

⟨*Evaluate BSDF at hit point*⟩ ≡ 43
```
BSDF *bsdf = isect.GetBSDF(ray, arena);
```

Figure 1.20 shows a few quantities that will be used frequently in the fragments to come; p represents the position of the ray-primitive intersection, and n is the surface normal at the intersection point. The normalized direction from the hit point back to the ray origin is stored in wo; because Cameras are responsible for normalizing the direction component of generated rays, there's no need to renormalize it here. Normalized directions are denoted by the ω symbol in this book, and in pbrt's code we will use the shorthand wo for ω_o, the outgoing direction of scattered light.

BSDF 478
BSDF::dgShading 479
Camera 302
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection::GetBSDF() 484
Intersection::Le() 625
Normal 65
Point 63
Ray::d 67
Ray::depth 67
Texture 519
Vector 57
WhittedIntegrator::
 maxDepth 42

⟨*Initialize common variables for Whitted integrator*⟩ ≡ 43
```
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Vector wo = -ray.d;
```

In case the ray happened to hit geometry that is emissive (such as an area light source), the integrator computes the emitted radiance by calling the Intersection::Le() method. This gives the first term of the light transport equation, Equation (1.1) on page 11. If the object is not emissive, this method returns a black spectrum.

⟨*Compute emitted light if ray hit an area light source*⟩ ≡ 43
```
L += isect.Le(wo);
```

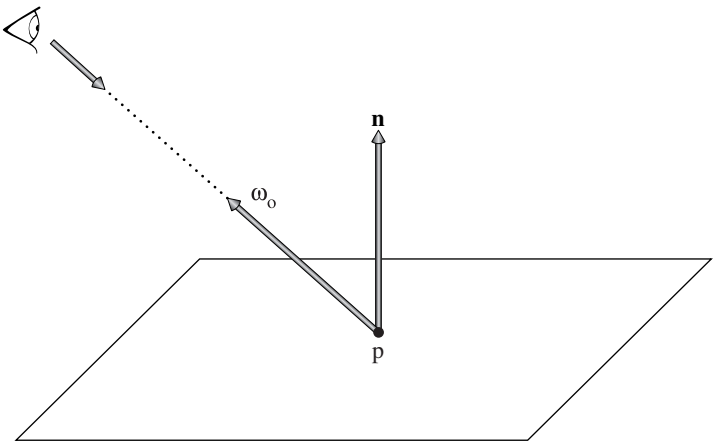


Figure 1.20: Geometric Setting for the Whitted Integrator. p is the ray intersection point and **n** is the surface normal there. The direction in which we'd like to compute reflected radiance is ω_o; it is the vector pointing in the opposite direction of the incident ray.

For each light, the integrator calls the `Light::Sample_L()` method to compute the radiance from that light falling on the surface at the point being shaded. This method also returns the direction vector from the point being shaded to the light source, which is stored in the variable `wi` (denoting an incident direction ω_i).

The spectrum returned by this method does not account for the possibility that some other shape may block light from the light and prevent it from reaching the point being shaded. Instead, it returns a `VisibilityTester` object that can be used to determine if any primitives block the surface point from the light source. This test is done by tracing a shadow ray between the point being shaded and the light to verify that the path is clear. `pbrt`'s code is organized in this way so that it can avoid tracing the shadow ray unless necessary. Specifically, this way it can first make sure that the light falling on the surface *would* be scattered in the direction ω_o if the light isn't blocked. For example, if the surface is not transmissive, then light arriving at the back side of the surface doesn't contribute to reflection.

The `Sample_L()` method also returns the probability density for the light to have sampled the direction `wi` in the `pdf` variable. This value is used for Monte Carlo integration with complex area light sources where light is arriving at the point from many directions even though just one direction is sampled here; for simple lights like point lights, the value of `pdf` is one. The details of how this probability density is computed and used in rendering are the topic of Chapters 13 to 15; in the end, the light's contribution must be divided by `pdf`, so this is done by the implementation here.

`Light::Sample_L()` 608

`VisibilityTester` 608

If the arriving radiance is nonzero and the BSDF indicates that some of the incident light from the direction ω_i is in fact scattered to the outgoing direction ω_o, then the integrator multiplies the radiance value L_i by the value of the BSDF f; the cosine term

(computed via the dot product between the ω_i vector and the surface normal); and the transmittance between the intersection point and the light. This product represents the light's contribution to the light transport equation integral, Equation (1.1), and it is added to the total reflected radiance L_o. After all lights have been considered, the integrator has computed the total contribution of *direct lighting*—light that arrives at the surface directly from emissive objects (as opposed to light that has reflected off other objects in the scene before arriving at the point).

⟨*Add contribution of each light source*⟩ ≡ 43

```
for (uint32_t i = 0; i < scene->lights.size(); ++i) {
    Vector wi;
    float pdf;
    VisibilityTester visibility;
    Spectrum Li = scene->lights[i]->Sample_L(p, isect.rayEpsilon,
        LightSample(rng), ray.time, &wi, &pdf, &visibility);
    if (Li.IsBlack() || pdf == 0.f) continue;
    Spectrum f = bsdf->f(wo, wi);
    if (!f.IsBlack() && visibility.Unoccluded(scene))
        L += f * Li * AbsDot(wi, n) *
             visibility.Transmittance(scene, renderer,
                                      sample, rng, arena) / pdf;
}
```

This integrator also handles light scattered by perfectly specular surfaces like mirrors or glass. It is fairly simple to use properties of mirrors to find the reflected directions (Figure 1.21), and to use Snell's law to find the transmitted directions (Section 8.2). The integrator can then recursively follow the appropriate ray in the new direction and add its contribution to the reflected radiance at the point originally seen from the camera. The computation of the effect of specular reflection and transmission is handled in separate utility functions so these functions can easily be reused by other `Integrators`.

AbsDot() 61
BSDF::f() 481
Integrator 740
Intersection::rayEpsilon 186
Light::Sample_L() 608
LightSample 710
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VisibilityTester 608
VisibilityTester::
Transmittance() 609
VisibilityTester::
Unoccluded() 609

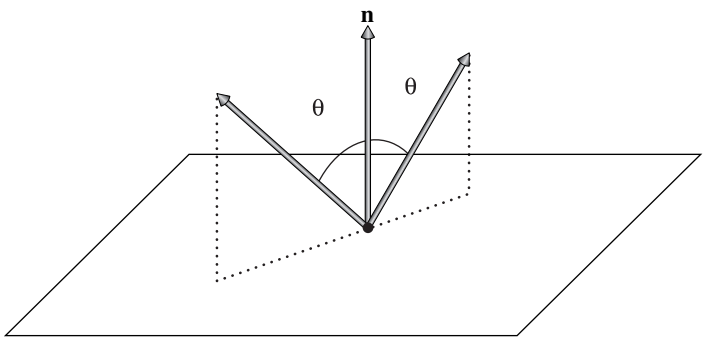


Figure 1.21: Reflected rays due to perfect specular reflection make the same angle with the surface normal as the incident ray.

⟨*Trace rays for specular reflection and refraction*⟩ ≡ 43, 789
```
L += SpecularReflect(ray, bsdf, rng, isect, renderer, scene, sample,
                     arena);
L += SpecularTransmit(ray, bsdf, rng, isect, renderer, scene, sample,
                      arena);
```

In the `SpecularReflect()` and `SpecularTransmit()` functions, the `BSDF::Samplef()` method returns an incident ray direction for a given outgoing direction and a given mode of light scattering. This method is one of the foundations of the Monte Carlo light transport algorithms that will be the subject of the last few chapters of this book. Here, we will use it to find only outgoing directions corresponding to perfect specular reflection or refraction, using flags to indicate to `BSDF::Sample_f()` that other types of reflection should be ignored. Although `BSDF::Sample_f()` can sample random directions leaving the surface for probabilistic integration algorithms, the randomness is constrained to be consistent with the `BSDF`'s scattering properties. In the case of perfect specular reflection or refraction, only one direction is possible, so there is no randomness at all.

The calls to `BSDF::Sample_f()` in these functions initialize `wi` with the chosen direction and return the BSDF's value for the directions (ω_o, ω_i). If the value of the BSDF is nonzero, the integrator uses the `Renderer::Li()` method to get the incoming radiance along ω_i, which in this case will in turn cause the `WhittedIntegrator::Li()` method to be called again. To compute the cosine term of the reflection integral, the integrator calls the `AbsDot()` function, which returns the absolute value of the dot product between two vectors. If the vectors are normalized, as both `wi` and `n` are here, this is equal to the absolute value of the cosine of the angle between them (Section 2.2.3).

Finally, in order to use ray differentials to antialias textures that are seen in reflections or refractions, it is necessary to know how reflection and transmission affect the screen-space footprint of rays. The fragments that compute the ray differentials for these rays are defined later, in Section 10.1.3.

⟨*Integrator Utility Functions*⟩ ≡
```
Spectrum SpecularReflect(const RayDifferential &ray, BSDF *bsdf,
        RNG &rng, const Intersection &isect, const Renderer *renderer,
        const Scene *scene, const Sample *sample, MemoryArena &arena) {
    Vector wo = -ray.d, wi;
    float pdf;
    const Point &p = bsdf->dgShading.p;
    const Normal &n = bsdf->dgShading.nn;
    Spectrum f = bsdf->Sample_f(wo, &wi, BSDFSample(rng), &pdf,
                                BxDFType(BSDF_REFLECTION | BSDF_SPECULAR));
    Spectrum L = 0.f;
    if (pdf > 0.f && !f.IsBlack() && AbsDot(wi, n) != 0.f) {
        ⟨Compute ray differential rd for specular reflection 512⟩
        Spectrum Li = renderer->Li(scene, rd, sample, rng, arena);
        L = f * Li * AbsDot(wi, n) / pdf;
    }
    return L;
}
```

AbsDot() 61
BSDF 478
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDFSample 705
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection 186
MemoryArena 1015
Normal 65
Point 63
Ray::d 67
RayDifferential 69
Renderer 24
Renderer::Li() 25
RNG 1003
Sample 343
Scene 22
Spectrum 263
Spectrum::IsBlack() 265
SpecularReflect() 46
SpecularTransmit() 47
Vector 57
WhittedIntegrator::Li() 42

The SpecularTransmit() function is essentially the same as SpecularReflect(), but just requests the BSDF_TRANSMISSION specular component of the BSDF, if any, rather than the BSDF_REFLECTION component used by SpecularReflect(). We therefore won't include its implementation in the text of the book here.

1.4 HOW TO PROCEED THROUGH THIS BOOK

We have written this book assuming it will be read in roughly front-to-back order. We have tried to minimize the number of forward references to ideas and interfaces that haven't yet been introduced, but do assume that the reader is acquainted with the previous content at any particular point in the text. However, some sections go into depth about advanced topics that some readers may wish to skip over (particularly on first reading); each advanced section is identified by an asterisk in its title.

Because of the modular nature of the system, the main requirement is that the reader be familiar with the low-level classes like Point, Ray, and Spectrum; the interfaces defined by the abstract base classes listed in Table 1.1; and the main rendering loop in SamplerRenderer::Render(). Given that knowledge, for example, the reader who doesn't care about precisely how a camera model based on a perspective projection matrix maps samples to rays can skip over the implementation of that camera and can just remember that the Camera::GenerateRayDifferential() method somehow turns a Sample into a RayDifferential.

The rest of this book is divided into four main parts of a few chapters each. First, Chapters 2 through 4 define the main geometric functionality in the system. Chapter 2 has the low-level classes like Point, Ray, and BBox. Chapter 3 defines the Shape interface, gives implementations of a number of shapes, and shows how to perform ray-shape intersection tests. Chapter 4 has the implementations of the acceleration structures for speeding up ray tracing by avoiding tests with primitives that a ray can be shown to definitely not intersect.

The second part covers the image formation process. First, Chapter 5 introduces the physical units used to measure light, and the Spectrum class that represents wavelength-varying distributions (i.e., color). Chapter 6 defines the Camera interface and has a few different camera implementations. The Sampler classes that place samples on the image plane are the topic of Chapter 7, and the overall process of turning radiance values on the film into images suitable for display is explained in Section 7.8.

The third part of the book is about light and how it scatters from surfaces and participating media. Chapter 8 includes a set of building-block classes that define a variety of types of reflection from surfaces. Materials, described in Chapter 9, use these reflection functions to implement a number of different surface materials, such as plastic, glass, and metal. Chapter 10 introduces texture, which describes variation in material properties (color, roughness, etc.) over surfaces, and Chapter 11 has the abstractions that describe how light is scattered and absorbed in participating media. Finally, Chapter 12 has the interface for light sources and light source implementations.

The last part brings all of the ideas from the rest of the book together to implement a number of light transport algorithms. Chapters 13 and 14 introduce the theory of

BBox 70
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
Camera 302
Camera::
GenerateRayDifferential() 303
Point 63
Ray 66
RayDifferential 69
Sample 343
Sampler 340
SamplerRenderer::Render() 27
Shape 108
Spectrum 263

Monte Carlo integration, a statistical technique for estimating the value of complex integrals, and have low-level routines for applying Monte Carlo to illumination and light scattering. The surface and volume integrators in Chapters 15 and 16 use Monte Carlo integration to compute more accurate approximations of the light transport equation than the `WhittedIntegrator`, using techniques like path tracing, irradiance caching, and photon mapping. Chapter 17 shows a number of ways to use `pbrt` for precomputing data to use for later rendering; this is an increasingly important topic as researchers continue to develop new techinques for efficiently re-rendering scenes for interactive applications using data computed offline with rendering systems like `pbrt`.

Finally, the last chapter of the book provides a brief retrospective and discussion of system design decisions along with a number of suggestions for more far-reaching projects than those in the exercises. Appendices describe utility functions and details of how the scene description is created as the input file is parsed.

1.4.1 THE EXERCISES

At the end of each chapter you will find exercises related to the material covered in that chapter. Each exercise is marked as one of three levels of difficulty:

1. An exercise that should take only an hour or two
2. A reading and/or implementation task that would be suitable for a course assignment and should take between 10 and 20 hours of work
3. A suggested final project for a course that will likely take 40 hours or more to complete

1.5 USING AND UNDERSTANDING THE CODE

We have written `pbrt` in C++. However, we have used only a subset of the language, both to make the code easy to understand, as well as to maximize the system's portability. In particular, we have avoided multiple inheritance and run-time exception handling and have used only a small subset of C++'s extensive standard library.

We will occasionally omit short sections of `pbrt`'s source code from this document. For example, when there are a number of cases to be handled, all with nearly identical code, we will present one case and note that the code for the remaining cases has been omitted from the text. Of course, all the omitted code can be found in the `pbrt` source code distribution.

1.5.1 POINTER OR REFERENCE?

C++ provides two different mechanisms for passing the address of a data structure to a function: pointers and references. If a function argument is not intended as an output variable, either can be used to save the expense of passing the entire structure on the stack. By convention, `pbrt` uses pointers when the function argument will be modified in some way, and `const` references when it won't. One important exception to this rule is that we will always use a pointer when we want to be able to pass `NULL` to indicate that a parameter is not available or should not be used.

WhittedIntegrator 42

1.5.2 CODE OPTIMIZATION

We have tried to make `pbrt` efficient through the use of well-chosen algorithms rather than through local micro-optimizations. However, we have applied some local optimizations to the parts of `pbrt` that account for the most execution time, as long as doing so didn't make the code confusing. There are two main local optimization principles used throughout the code:

- On current CPU architectures, the slowest mathematical operations are divides, square roots, and trigonometric functions. Addition, subtraction, and multiplication are generally 10 to 50 times faster than those operations. Reducing the number of slow mathematical operations can help performance substantially; for example, instead of repeatedly dividing by some value v, we will tend to precompute the reciprocal $1/v$ and multiply by that instead.
- The speed of CPUs continues to grow more quickly than the speed at which data can be loaded from main memory into the CPU. This means that waiting for values to be fetched from memory can be a major performance limitation. Organizing algorithms and data structures in ways that give good performance from memory caches can speed up program execution much more than reducing the total number of instructions executed. Section A.5 in Appendix A discusses general principles for memory-efficient programming; these ideas are mostly applied in the ray intersection acceleration structures of Chapter 4 and the image map representation in Section 10.4.2, although they influence many of the design decisions throughout the system.

1.5.3 THE BOOK WEB SITE

We have created a companion Web site for this book, located at *www.pbrt.org*. The Web site includes the system's complete source code, additional modules, example scenes, and errata. It also hosts contributed `pbrt` modules and scene file conversion tools from readers.

All readers are encouraged to visit the Web site and follow the instructions to subscribe to the *pbrt-announce* mailing list; information about how to join it is available at *pbrt.org/lists.php*. We will occasionally send announcements of software updates to this list, so readers can always have the latest version of `pbrt` and all the latest modules. The Web site also has links to an online discussion forum for `pbrt` and a bug-tracking system.

1.5.4 EXTENDING THE SYSTEM

One of our goals in writing this book and building the `pbrt` system was to make it easier for developers and researchers to experiment with new (or old!) ideas in rendering. One of the great joys in computer graphics is writing new software that makes a new image; even small changes to the system can be fun to experiment with. The exercises throughout the book suggest many changes to make to the system, ranging from small tweaks to major open-ended research projects. Section B.4 in Appendix B has more information about the mechanics of how to extend the system.

1.5.5 BUGS

Although we have made every effort to make pbrt as correct as possible through extensive testing, it is inevitable that some bugs are still present.

If you believe you have found a bug in the system, please do the following:

1. Reproduce the bug with an unmodified copy of the latest version of pbrt.
2. Check the online discussion forum and the bug-tracking system at *www.pbrt.org*. Your issue may be a known bug, or it may be a commonly misunderstood feature.
3. Try to find make the simplest possible test case that demonstrates the bug. Many bugs can be demonstrated by input files that are just a few lines long, and debugging is much easier with a simple scene than a complex one.
4. Submit a detailed bug report using our online bug-tracking system. Make sure that you include the short input file that demonstrates the bug and a detailed description of why you think pbrt is not behaving correctly on your input. If you can provide a patch to the code that fixes the bug, all the better!

We will periodically release updated versions of pbrt with bug fixes and minor enhancements. However, we will not make major changes to the updated versions of the pbrt source code distributed from the book's Web site so that the system described here in the book doesn't diverge excessively from the updated versions of pbrt.

FURTHER READING

In a seminal early paper, Arthur Appel (1968) first described the basic idea of ray tracing to solve the hidden surface problem and to compute shadows in polygonal scenes. Goldstein and Nagel (1971) later showed how ray tracing could be used to render scenes with quadric surfaces. Kay and Greenberg (1979) described a ray-tracing approach to rendering transparency, and Whitted's seminal *CACM* article described the general recursive ray-tracing algorithm we have outlined in this chapter, accurately simulating reflection and refraction from specular surfaces and shadows from point light sources (Whitted 1980). Heckbert (1987) was the first to explore realistic rendering of dessert.

Notable early books on physically based rendering and image synthesis include Cohen and Wallace's *Radiosity and Realistic Image Synthesis* (Cohen and Wallace 1993), Sillion and Puech's *Radiosity and Global Illumination* (Sillion and Puech 1994), and Ashdown's *Radiosity: A Programmer's Perspective* (Ashdown 1994), all of which primarily describe the finite-element radiosity method. Glassner's *Principles of Digital Image Synthesis* (Glassner 1995) is an encyclopedic two-volume summary of theoretical foundations for realistic rendering. Hall's *Illumination and Color in Computer Generated Imagery* (Hall 1989) is one of the first books to present rendering in a physically based framework. Dutré et al.'s *Advanced Global Illumination* has extensive and up-to-date coverage of these topics (Dutré, Bala, and Bekaert 2006).

Greenberg et al. (1997) made a strong argument for a physically accurate rendering based on measurements of the material properties of real-world objects and on deep understanding of the human visual system. These ideas have served as a basis for much of the rendering research done at Cornell over the past two decades.

In a paper on ray-tracing system design, Kirk and Arvo (1988) suggested many principles that have now become classic in renderer design. Their renderer was implemented as a core kernel that encapsulated the basic rendering algorithms and interacted with primitives and shading routines via a carefully constructed object-oriented interface. This approach made it easy to extend the system with new primitives and acceleration methods. `pbrt`'s design is based on these ideas.

Another good reference on ray tracer design is *Introduction to Ray Tracing* (Glassner 1989a), which describes the state of the art in ray tracing at that time and has a chapter by Heckbert that sketches the design of a basic ray tracer. More recently, Shirley and Morley's *Realistic Ray Tracing* gives an easy-to-understand introduction to ray tracing and includes the complete source code to a basic ray tracer (Shirley and Morley 2008). Suffern's book also provides a gentle introduction to ray tracing (Suffern 2007).

Researchers at Cornell University have developed a rendering testbed over many years; its design and overall structure were described by Trumbore, Lytle, and Greenberg (1993). Its predecessor was described by Hall and Greenberg (1983). This system is a loosely coupled set of modules and libraries, each designed to handle a single task (ray–object intersection acceleration, image storage, etc.) and written in a way that makes it easy to combine appropriate modules to investigate and develop new rendering algorithms. This testbed has been quite successful, serving as the foundation for much of the rendering research done at Cornell.

Many papers have been written that describe the design and implementation of other rendering systems, including renderers for entertainment and artistic applications. The Reyes architecture, which forms the basis for Pixar's RenderMan® renderer, was first described by Cook, Carpenter, and Catmull (1987), and a number of improvements to the original algorithm have been summarized by Apodaca and Gritz (2000). Gritz and Hahn (1996) described the BMRT ray tracer. The renderer in the Maya modeling and animation system was described by Sung et al. (1998), and some of the internal structure of the *mental ray* renderer is described in Driemeyer and Herken's book on its API (Driemeyer and Herken 2002). The design of the high-performance *Manta* interactive ray tracer is described by Bigler et al. (2006).

Another category of renderer focuses on physically based rendering, like `pbrt`. One of the first renderers based fundamentally on physical quantities is called *Radiance*, and it has been used widely in lighting simulation applications. Ward described its design and history in a paper and a book (Ward 1994b; Larson and Shakespeare 1998). *Radiance* is designed in the UNIX style, as a set of interacting programs, each handling a different part of the rendering process. This general type of rendering architecture was first described by Duff (1985).

Glassner's *Spectrum* rendering architecture also focuses on physically based rendering (Glassner 1993), approached through a signal-processing-based formulation of the problem. It is an extensible system built with a plug-in architecture; `pbrt`'s approach of using parameter/value lists for initializing implementations of the main abstract interfaces is similar to *Spectrum*'s. One notable feature of *Spectrum* is that all parameters that describe the scene can be functions of time.

Slusallek and Seidel described the architecture of the *Vision* rendering system, which is also physically based and designed to support a wide variety of light transport algorithms (Slusallek and Seidel 1995, 1996; Slusallek 1996). In particular, it has the ambitious goal of supporting both Monte Carlo and finite-element-based light transport algorithms.

The source code to `pbrt` is licensed under the GNU General Public License; this has made it possible for other developers to use `pbrt` code as a basis for their efforts. *LuxRender*, available from *www.luxrender.net*, is a physically based renderer built using `pbrt` as a starting point; it offers a number of additional features and has a rich set of scene export plugins for modeling systems.

The source code to a number of other ray tracers and renderers is available on the Web. Because this is a rapidly changing field, we maintain a set of links to interesting open source rendering systems online at *www.pbrt.org/links.php*.

A good introduction to the C++ programming language and C++ standard library is the third edition of Stroustrup's *The C++ Programming Language* (Stroustrup 1997).

EXERCISE

1 1.1 A good way to gain an understanding of `pbrt` is to follow the process of computing the radiance value for a single ray in a debugger. Build a version of `pbrt` with debugging symbols and set up your debugger to run `pbrt` with the `killeroo-simple.pbrt` scene from the `scenes/` directory. Set breakpoints in the `SamplerRenderer::Render()` and `SamplerRendererTask::Run()` methods and trace through the process of how a ray is generated, how its radiance value is computed, and how its contribution is added to the image. The first time you do this, you may want to specify that only a single thread of execution should be used by providing `--ncores 1` as command-line arguments to `pbrt`; doing so ensures that all computation is done in the main processing thread, which may make it easier to understand what is going on, depending on how easy your debugger makes it to step through the program when it is running multiple threads.

As you gain more understanding about the details of the system later in the book, repeat this process and trace through particular parts of the system more carefully.

`SamplerRenderer::Render()` 27
`SamplerRendererTask::Run()` 30

This page intentionally left blank



CHAPTER TWO

02 GEOMETRY AND TRANSFORMATIONS

Almost all nontrivial graphics programs are built on a foundation of geometric classes. These classes represent mathematical constructs like points, vectors, and rays. Because these classes are ubiquitous throughout the system, good abstractions and efficient implementations are critical. This chapter presents the interface to and implementation of `pbrt`'s geometric foundation. Note that these are not the classes that represent the actual scene geometry (triangles, spheres, etc.); those classes are the topic of Chapter 3.

The geometric classes in this chapter are defined in the files `core/geometry.h` and `core/geometry.cpp` in the `pbrt` distribution and the implementation of transformation matrices (Section 2.7) are in the files `core/transform.h` and `core/transform.cpp`.

2.1 COORDINATE SYSTEMS

As is typical in computer graphics, `pbrt` represents three-dimensional points, vectors, and normal vectors with three floating-point coordinate values: x, y, and z. Of course, these values are meaningless without a *coordinate system* that defines the origin of the space and gives three linearly independent vectors that define the x, y, and z axes of the space. Together, the origin and three vectors are called the *frame* that defines the coordinate system. Given an arbitrary point or direction in 3D, its (x, y, z) coordinate values depend on its relationship to the frame. Figure 2.1 shows an example that illustrates this idea in 2D.

In the general n-dimensional case, a frame's origin p_o and its n linearly independent basis vectors define an n-dimensional *affine space*. All vectors $\mathbf{v}$ in the space can be expressed as a linear combination of the basis vectors. Given a vector $\mathbf{v}$ and the basis vectors $\mathbf{v}_i$,

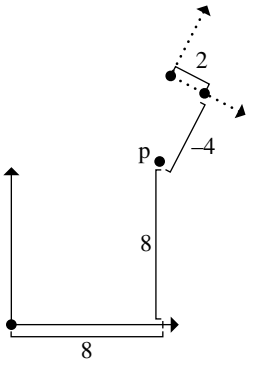


Figure 2.1: In 2D, the (x, y) coordinates of a point p are defined by the relationship of the point to a particular 2D coordinate system. Here, two coordinate systems are shown; the point might have coordinates $(8, 8)$ with respect to the coordinate system with its coordinate axes drawn in solid lines, but have coordinates $(2, -4)$ with respect to the coordinate system with dashed axes. In either case, the 2D point p is at the same absolute position in space.

there is a unique set of scalar values s_i such that

$$\mathbf{v} = s_1\mathbf{v}_1 + \cdots + s_n\mathbf{v}_n.$$

The scalars s_i are the *representation* of $\mathbf{v}$ with respect to the basis $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ and are the coordinate values that we store with the vector. Similarly, for all points p, there are unique scalars s_i such that the point can be expressed in terms of the origin p_o and the basis vectors

$$\mathrm{p} = \mathrm{p}_o + s_1\mathbf{v}_1 + \cdots + s_n\mathbf{v}_n.$$

Thus, although points and vectors are both represented by x, y, and z coordinates in 3D, they are distinct mathematical entities and are not freely interchangeable.

This definition of points and vectors in terms of coordinate systems reveals a paradox: to define a frame we need a point and a set of vectors, but we can only meaningfully talk about points and vectors with respect to a particular frame. Therefore, in three dimensions we need a *standard frame* with origin (0, 0, 0) and basis vectors (1, 0, 0), (0, 1, 0), and (0, 0, 1). All other frames will be defined with respect to this canonical coordinate system which we call *world space*.

2.1.1 COORDINATE SYSTEM HANDEDNESS

There are two different ways that the three coordinate axes can be arranged, as shown in Figure 2.2. Given perpendicular x and y coordinate axes, the z axis can point in one of two directions. These two choices are called *left-handed* and *right-handed*. The choice between the two is arbitrary, but has a number of implications for how some of the geometric operations throughout the chapter are defined. `pbrt` uses a left-handed coordinate system.

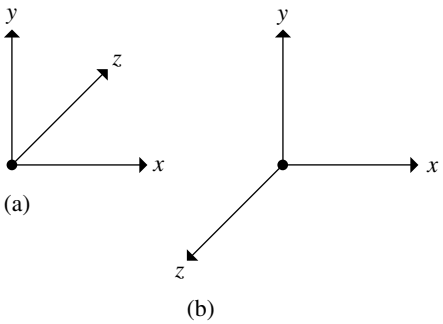


Figure 2.2: (a) In a left-handed coordinate system, the z axis points into the page when the x and y axes are oriented with x pointing to the right and y pointing up. (b) In a right-handed system, the z axis points out of the page.

2.2 VECTORS

⟨*Geometry Declarations*⟩ ≡
```
class Vector {
public:
    ⟨Vector Public Methods 58⟩
    ⟨Vector Public Data 57⟩
};
```

A `Vector` in `pbrt` represents a direction in 3D space. As described earlier, vectors are represented with a three-tuple of components that give its representation in terms of the x, y, and z axes of the space it is defined in. The individual components of a vector $\mathbf{v}$ will be written $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$.

⟨*Vector Public Data*⟩ ≡ 57
```
float x, y, z;
```

Readers who are experienced in object-oriented design might object to our decision to make the `Vector` data publicly accessible. Typically, data members are only accessible inside the class, and external code that wishes to access or modify the contents of a class must do so through a well-defined API of selector and mutator functions. Although we generally agree with this design principle, it is not appropriate here. The purpose of selector and mutator functions is to hide the class's internal implementation details. In the case of `Vectors`, hiding this basic part of their design gains nothing and adds bulk to the class usage.

`Assert()` 1005
`Vector` 57

By default, the (x, y, z) values are set to zero, although the user of the class can optionally supply values for each of the components. If the user does supply values, we check that none of them has the floating-point not-a-number (NaN) value using the `Assert()` macro. When compiled in optimized mode, this macro disappears from the compiled code, saving the expense of verifying this case. `Assert()` is defined in Section A.3.1.

NaNs almost certainly indicate a bug in the system; if NaNs are generated by some computation, we'd like to catch them as soon as possible in order to make isolating their source easier.

⟨*Vector Public Methods*⟩ ≡ 57
```
Vector() { x = y = z = 0.f; }
Vector(float xx, float yy, float zz)
    : x(xx), y(yy), z(zz) {
    Assert(!HasNaNs());
}
```

The code to check for NaNs just calls the standard math library `isnan()` function on each of x, y, and z.

⟨*Vector Public Methods*⟩ +≡ 57
```
bool HasNaNs() const { return isnan(x) || isnan(y) || isnan(z); }
```

2.2.1 ARITHMETIC

Addition and subtraction of vectors are done component-wise. The usual geometric interpretation of vector addition and subtraction is shown in Figures 2.3 and 2.4.

⟨*Vector Public Methods*⟩ +≡ 57
```
Vector operator+(const Vector &v) const {
    return Vector(x + v.x, y + v.y, z + v.z);
}

Vector& operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

The code for subtracting two vectors is similar, and therefore not shown here.

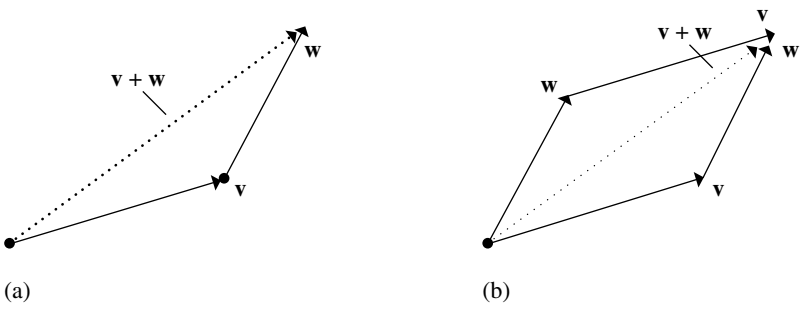


Figure 2.3: (a) Vector addition: $\mathbf{v} + \mathbf{w}$. (b) Notice that the sum $\mathbf{v} + \mathbf{w}$ forms the diagonal of the parallelogram formed by $\mathbf{v}$ and $\mathbf{w}$, which shows the commutativity of vector addition: $\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}$.

Assert() 1005
Vector 57
Vector::HasNaNs() 58

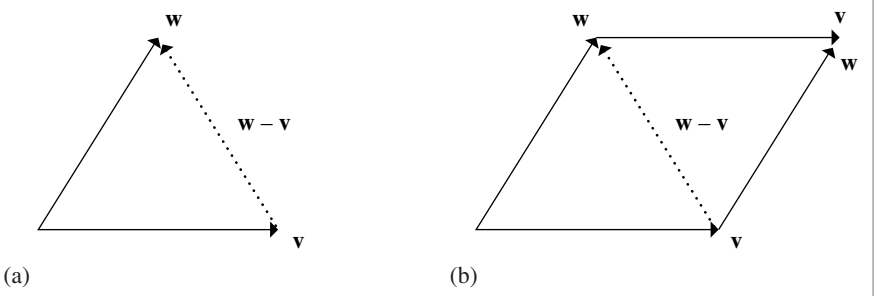


Figure 2.4: (a) Vector subtraction. (b) The difference $-\mathbf{v} - \mathbf{w}$ is the other diagonal of the parallelogram formed by $\mathbf{v}$ and $\mathbf{w}$.

2.2.2 SCALING

A vector can be multiplied component-wise by a scalar, thereby changing its length. Three functions are needed in order to cover all of the different ways that this operation may be written in source code (i.e., `v*s`, `s*v`, and `v *= s`):

⟨*Vector Public Methods*⟩ +≡ 57

```
Vector operator*(float f) const { return Vector(f*x, f*y, f*z); }

Vector &operator*=(float f) {
    x *= f; y *= f; z *= f;
    return *this;
}
```

⟨*Geometry Inline Functions*⟩ ≡

```
inline Vector operator*(float f, const Vector &v) { return v*f; }
```

Similarly, a vector can be divided component-wise by a scalar. The code for scalar division is similar to scalar multiplication, although division of a scalar by a vector is not well defined and so is not permitted.

In the implementation of these methods, we use a single division to compute the scalar's reciprocal, then perform three component-wise multiplications. This is a useful trick for avoiding division operations, which are generally much slower than multiplies on modern CPUs.[1]

1 It is a common misconception that these sorts of optimizations are unnecessary because the compiler will perform the necessary analysis. Compilers are frequently restricted from performing many optimizations of this type. For example, given two floating-point numbers, the quantities $a + b$ and $b + a$ are not candidates for common subexpression elimination because the IEEE floating-point standard (which is standard on modern architectures) does not guarantee that the two sums will be identical. In fact, compilers are limited in their ability to legally make these substitutions because expressions may have been carefully ordered in order to minimize round-off error. For division, IEEE requires that $x/x = 1$ for all x, but if we compute $1/x$ and store it in a variable and then multiply x by that value, it is not guaranteed that 1 will be the result. In this case, we are willing to lose that guarantee in exchange for higher performance.

Vector 57

We use the `Assert()` macro to make sure that the given scale is not zero; this should never happen and would indicate a bug elsewhere in the system.

⟨*Vector Public Methods*⟩ +≡ 57
```
Vector operator/(float f) const {
    Assert(f != 0);
    float inv = 1.f / f;
    return Vector(x * inv, y * inv, z * inv);
}

Vector &operator/=(float f) {
    Assert(f != 0);
    float inv = 1.f / f;
    x *= inv; y *= inv; z *= inv;
    return *this;
}
```

The `Vector` class also provides a unary negation operator that returns a new vector pointing in the opposite direction of the original one:

⟨*Vector Public Methods*⟩ +≡ 57
```
Vector operator-() const { return Vector(-x, -y, -z); }
```

Some routines will find it useful to be able to easily loop over the components of a `Vector`; the `Vector` class also provides a C++ operator so that, given a vector `v`, `v[0] == v.x` and so forth.

⟨*Vector Public Methods*⟩ +≡ 57
```
float operator[](int i) const {
    Assert(i >= 0 && i <= 2);
    return (&x)[i];
}

float &operator[](int i) {
    Assert(i >= 0 && i <= 2);
    return (&x)[i];
}
```

2.2.3 DOT AND CROSS PRODUCT

Two useful operations on vectors are the dot product (also known as the scalar or inner product) and the cross product. For two vectors $\mathbf{v}$ and $\mathbf{w}$, their *dot product* $(\mathbf{v} \cdot \mathbf{w})$ is defined as:

$$\mathbf{v}_x\mathbf{w}_x + \mathbf{v}_y\mathbf{w}_y + \mathbf{v}_z\mathbf{w}_z.$$

⟨*Geometry Inline Functions*⟩ +≡
```
inline float Dot(const Vector &v1, const Vector &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

Assert() 1005
Vector 57

The dot product has a simple relationship to the angle between the two vectors:

$$(\mathbf{v} \cdot \mathbf{w}) = \|\mathbf{v}\| \, \|\mathbf{w}\| \cos\theta, \tag{2.1}$$

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$, and $\|\mathbf{v}\|$ denotes the length of the vector $\mathbf{v}$. It follows from this that $(\mathbf{v} \cdot \mathbf{w})$ is zero if and only if $\mathbf{v}$ and $\mathbf{w}$ are perpendicular, provided that neither $\mathbf{v}$ nor $\mathbf{w}$ is *degenerate*—equal to (0, 0, 0). A set of two or more mutually perpendicular vectors is said to be *orthogonal*. An orthogonal set of unit vectors is called *orthonormal*.

It immediately follows from Equation (2.1) that if $\mathbf{v}$ and $\mathbf{w}$ are unit vectors, their dot product is exactly the cosine of the angle between them. As the cosine of the angle between two vectors often needs to be computed in computer graphics, we will frequently make use of this property.

A few basic properties directly follow from the definition. For example, if $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are vectors and s is a scalar value, then:

$$(\mathbf{u} \cdot \mathbf{v}) = (\mathbf{v} \cdot \mathbf{u})$$
$$(s\mathbf{u} \cdot \mathbf{v}) = s(\mathbf{v} \cdot \mathbf{u})$$
$$(\mathbf{u} \cdot (\mathbf{v} + \mathbf{w})) = (\mathbf{u} \cdot \mathbf{v}) + (\mathbf{u} \cdot \mathbf{w}).$$

We will frequently need to compute the absolute value of the dot product as well. The `AbsDot()` function does this for us so that we don't need a separate call to `fabsf()`:

⟨*Geometry Inline Functions*⟩ +≡
```
inline float AbsDot(const Vector &v1, const Vector &v2) {
    return fabsf(Dot(v1, v2));
}
```

The *cross product* is another useful vector operation. Given two vectors in 3D, the cross product $\mathbf{v} \times \mathbf{w}$ is a vector that is perpendicular to both of them. Note that this new vector can point in one of two directions; the coordinate system's handedness determines which is appropriate. Given orthogonal vectors $\mathbf{v}$ and $\mathbf{w}$, then $\mathbf{v} \times \mathbf{w}$ is defined to be a vector such that $(\mathbf{v}, \mathbf{w}, \mathbf{v} \times \mathbf{w})$ form a coordinate system of the appropriate handedness.

In a left-handed coordinate system, the cross product is defined as:

$$(\mathbf{v} \times \mathbf{w})_x = \mathbf{v}_y \mathbf{w}_z - \mathbf{v}_z \mathbf{w}_y$$
$$(\mathbf{v} \times \mathbf{w})_y = \mathbf{v}_z \mathbf{w}_x - \mathbf{v}_x \mathbf{w}_z$$
$$(\mathbf{v} \times \mathbf{w})_z = \mathbf{v}_x \mathbf{w}_y - \mathbf{v}_y \mathbf{w}_x.$$

An easy way to remember this is to compute the determinant of the matrix:

$$\mathbf{v} \times \mathbf{w} = \begin{vmatrix} i & j & k \\ \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z \\ \mathbf{w}_x & \mathbf{w}_y & \mathbf{w}_z \end{vmatrix},$$

where i, j, and k represent the axes (1, 0, 0), (0, 1, 0), and (0, 0, 1), respectively. Note that this equation is merely a memory aid and not a rigorous mathematical construction, since the matrix entries are a mix of scalars and vectors.

AbsDot() 61
Dot() 60
Vector 57

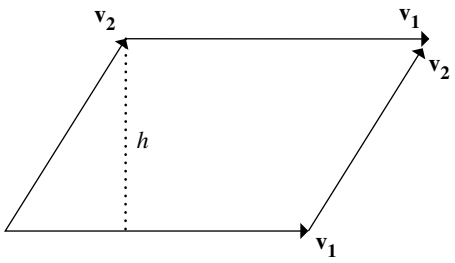


Figure 2.5: The area of a parallelogram with edges given by vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is equal to $\mathbf{v}_1 h$. From Equation (2.2), the length of the cross product of $\mathbf{v}_1$ and $\mathbf{v}_2$ is equal to the product of the two vector lengths times the sine of the angle between them—the parallelogram area.

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector Cross(const Vector &v1, const Vector &v2) {
    return Vector((v1.y * v2.z) - (v1.z * v2.y),
                  (v1.z * v2.x) - (v1.x * v2.z),
                  (v1.x * v2.y) - (v1.y * v2.x));
}
```

From the definition of the cross product, we can derive

$$\|\mathbf{v}\times\mathbf{w}\| = \|\mathbf{v}\|\,\|\mathbf{w}\|\sin\theta, \tag{2.2}$$

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$. An important implication of this is that the cross product of two perpendicular unit vectors is itself a unit vector. Note also that the result of the cross product is a degenerate vector if $\mathbf{v}$ and $\mathbf{w}$ are parallel.

This definition also shows a convenient way to compute the area of a parallelogram (Figure 2.5). If the two edges of the parallelogram are given by vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, and it has height h, the area is given by $\|\mathbf{v}_1\|h$. Since $h = \sin\theta\|\mathbf{v}_2\|$, we can use Equation (2.2) to see that the area is $\|\mathbf{v}_1\times\mathbf{v}_2\|$.

2.2.4 NORMALIZATION

It is often necessary to *normalize* a vector—that is, to compute a new vector pointing in the same direction but with unit length. A normalized vector is often called a *unit vector*. The function to do this is called `Vector::Normalize()`. The notation used in this book for normalized vectors is that $\hat{\mathbf{v}}$ is the normalized version of $\mathbf{v}$. `Normalize()` divides each component by the length of the vector, $\|\mathbf{v}\|$.

⟨*Vector Public Methods*⟩ +≡ 57
```
float LengthSquared() const { return x*x + y*y + z*z; }
float Length() const { return sqrtf(LengthSquared()); }
```

Vector 57
Vector::Normalize() 63

`Normalize()` returns a new vector; it does *not* normalize the vector in place:

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector Normalize(const Vector &v) { return v / v.Length(); }
```

2.2.5 COORDINATE SYSTEM FROM A VECTOR

We will frequently want to construct a local coordinate system given only a single vector. Because the cross product of two vectors is orthogonal to both, we can apply the cross product two times to get a set of three orthogonal vectors for the coordinate system. Note that the two vectors generated by this technique are unique only up to a rotation about the given vector.

The implementation of this function assumes that the vector passed in, `v1`, has already been normalized. It first constructs a perpendicular vector by zeroing one of the components of the original vector, swapping the remaining two, and negating one of them. Inspection of the two cases should make clear that `v2` will be normalized and that the dot product ($\mathbf{v}_1 \cdot \mathbf{v}_2$) must be equal to zero. Given these two perpendicular vectors, a single cross product gives the third, which by definition will be perpendicular to the first two.

⟨*Geometry Inline Functions*⟩ +≡
```
inline void CoordinateSystem(const Vector &v1, Vector *v2, Vector *v3) {
    if (fabsf(v1.x) > fabsf(v1.y)) {
        float invLen = 1.f / sqrtf(v1.x*v1.x + v1.z*v1.z);
        *v2 = Vector(-v1.z * invLen, 0.f, v1.x * invLen);
    }
    else {
        float invLen = 1.f / sqrtf(v1.y*v1.y + v1.z*v1.z);
        *v2 = Vector(0.f, v1.z * invLen, -v1.y * invLen);
    }
    *v3 = Cross(v1, *v2);
}
```

2.3 POINTS

⟨*Geometry Declarations*⟩ +≡
```
class Point {
public:
    ⟨Point Public Methods 64⟩
    ⟨Point Public Data 63⟩
};
```

A point is a zero-dimensional location in 3D space. The `Point` class in `pbrt` represents points in the obvious way: using x, y, and z coordinates with respect to their coordinate system. Although the same (x, y, z) representation is used for vectors, the fact that a point represents a position, whereas a vector represents a direction, leads to a number of important differences in how they are treated. Points are denoted in text by p.

Cross() 62
Point 63
Vector 57

⟨*Point Public Data*⟩ ≡ 63
```
float x, y, z;
```

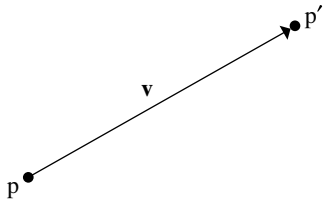


Figure 2.6: Obtaining the Vector between Two Points. The vector $p' - p$ is given by the component-wise subtraction of the points p' and p.

Like the `Vector` constructor, the `Point` constructor takes optional parameters to set the `x`, `y`, and `z` coordinate values.

⟨*Point Public Methods*⟩ ≡ 63
```
Point() { x = y = z = 0.f; }
Point(float xx, float yy, float zz)
    : x(xx), y(yy), z(zz) {
}
```

There are certain `Point` methods that either return or take a `Vector`. For instance, one can add a vector to a point, offsetting it in the given direction and obtaining a new point. Alternately, one can subtract one point from another, obtaining the vector between them, as shown in Figure 2.6.

⟨*Point Public Methods*⟩ +≡ 63
```
Point operator+(const Vector &v) const {
    return Point(x + v.x, y + v.y, z + v.z);
}

Point &operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

⟨*Point Public Methods*⟩ +≡ 63
```
Vector operator-(const Point &p) const {
    return Vector(x - p.x, y - p.y, z - p.z);
}

Point operator-(const Vector &v) const {
    return Point(x - v.x, y - v.y, z - v.z);
}
```

Point 63
Vector 57

```
Point &operator-=(const Vector &v) {
    x -= v.x; y -= v.y; z -= v.z;
    return *this;
}
```

The distance between two points is easily computed by subtracting them to compute the vector between them and then finding the length of that vector:

⟨*Geometry Inline Functions*⟩ +≡
```
inline float Distance(const Point &p1, const Point &p2) {
    return (p1 - p2).Length();
}
inline float DistanceSquared(const Point &p1, const Point &p2) {
    return (p1 - p2).LengthSquared();
}
```

Although in general it doesn't make sense mathematically to weight points by a scalar or add two points together, the `Point` class still allows these operations in order to be able to compute weighted sums of points, which is mathematically meaningful as long as the weights used all sum to one. The code for scalar multiplication and addition with `Point`s is identical to `Vector`s, so it is not shown here.

2.4 NORMALS

⟨*Geometry Declarations*⟩ +≡
```
class Normal {
public:
    ⟨Normal Public Methods 66⟩
    ⟨Normal Public Data⟩
};
```

A *surface normal* (or just *normal*) is a vector that is perpendicular to a surface at a particular position. It can be defined as the cross product of any two nonparallel vectors that are tangent to the surface at a point. Although normals are superficially similar to vectors, it is important to distinguish between the two of them: because normals are defined in terms of their relationship to a particular surface, they behave differently than vectors in some situations, particularly when applying transformations. This difference is discussed in Section 2.8.

The implementations of `Normal`s and `Vector`s are very similar. Like vectors, normals are represented by three `float`s `x`, `y`, and `z`; they can be added and subtracted to compute new normals; and they can be scaled and normalized. However, a normal cannot be added to a point, and one cannot take the cross product of two normals. Note that, in an unfortunate turn of terminology, normals are *not* necessarily normalized.

`Normal` 65
`Point` 63
`Vector` 57

The `Normal` provides an extra constructor that initializes a `Normal` from a `Vector`. Because `Normal`s and `Vector`s are different in subtle ways, we want to make sure that this conversion doesn't happen when we don't intend it to, but we'd like it to be possible where it is appropriate. Fortunately, the C++ `explicit` keyword ensures that conversion between

two compatible types only happens when the programmer explicitly requests such a conversion. `Vector` also provides a constructor that converts the other way. Thus, given the declarations `Vector v; Normal n;`, the assignment `n = v` is illegal, so it is necessary to explicitly convert the vector, as in `n = Normal(v)`.

⟨*Normal Public Methods*⟩ ≡ 65
```
explicit Normal(const Vector &v)
  : x(v.x), y(v.y), z(v.z) {
}
```

⟨*Vector Public Methods*⟩ +≡ 57
```
explicit Vector(const Normal &n);
```

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector::Vector(const Normal &n)
  : x(n.x), y(n.y), z(n.z) {
}
```

The `Dot()` and `AbsDot()` functions are also overloaded to compute dot products between the various possible combinations of normals and vectors. This code won't be included in the text here. We also won't include implementations of all of the various other `Normal` methods here, since they are similar to those for vectors.

It's often necessary to flip a surface normal so that it lies in the same hemisphere as a given vector—for example, the surface normal that lies in the same hemisphere as an outgoing ray is frequently needed. The `Faceforward()` utility function encapsulates this small computation. `pbrt` also provides variants of this function for the other three combinations of `Vectors` and `Normals` as parameters. Be careful when using the other instances, though: when using the version that takes two `Vectors`, for example, ensure that the first parameter is the one that should be returned (possibly flipped) and the second is the one to test against. Reversing the two parameters will give unexpected results.

⟨*Geometry Inline Functions*⟩ +≡
```
inline Normal Faceforward(const Normal &n, const Vector &v) {
    return (Dot(n, v) < 0.f) ? -n : n;
}
```

2.5 RAYS

⟨*Geometry Declarations*⟩ +≡
```
class Ray {
public:
    ⟨Ray Public Methods 68⟩
    ⟨Ray Public Data 67⟩
};
```

A *ray* is a semi-infinite line specified by its origin and direction. `pbrt` represents a `Ray` with a `Point` for the origin and a `Vector` for the direction. A ray is denoted by r; it has origin o and direction **d**, as shown in Figure 2.7. Because we will be referring to these variables

AbsDot() 61
Dot() 60
Normal 65
Point 63
Ray 66
Vector 57

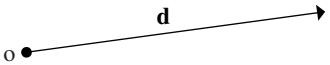


Figure 2.7: A ray is a semi-infinite line defined by its origin o and its direction vector **d**.

often throughout the code, the origin and direction members of a Ray are named o and d. Note that we again choose to make the data publicly available for convenience.

⟨*Ray Public Data*⟩ ≡ 66
```
Point o;
Vector d;
```

The *parametric form* of a ray expresses it as a function of a scalar value t, giving the set of points that the ray passes through:

$$\mathrm{r}(t) = \mathrm{o} + t\mathbf{d} \qquad 0 \leq t \leq \infty. \tag{2.3}$$

The Ray also includes fields to limit the ray to a particular segment along its infinite extent. These fields, called mint and maxt, allow us to restrict the ray to a segment of points [r(mint), r(maxt)]. Notice that these fields are declared as mutable, meaning that they can be changed even if the Ray structure that contains them is const.

Thus, when a ray is passed to a method that takes a const Ray &, that method is not allowed to modify its origin or direction but can modify its parametric range. This convention fits one of the most common uses of rays in the system, as parameters to ray–object intersection testing routines, which will record the offsets to the closest intersection in maxt.

⟨*Ray Public Data*⟩ +≡ 66
```
mutable float mint, maxt;
```

For simulating motion blur in scenes with animated objects, each ray may have a time value associated with it, so a data member to store time is available as well. The rendering system is responsible for constructing a representation of the scene at the appropriate time for each ray.

⟨*Ray Public Data*⟩ +≡ 66
```
float time;
```

In many light transport algorithms, a path of multiple rays is followed through the scene. A light path might start at a light source and scatter off of a few surfaces before it reached the camera, for example. The ray depth member variable allows light transport algorithms to track how many times light has bounced along the current path so that they can terminate the path after a particular number of bounces, etc.

Point 63
Ray 66
Vector 57

⟨*Ray Public Data*⟩ +≡ 66
```
int depth;
```

Constructing Rays is straightforward. The default constructor relies on the Point and Vector constructors to set the origin and direction to (0, 0, 0). Alternately, a particular point and direction can be provided. If an origin and direction are provided, the constructor requires that a value be given for mint as well (but maxt is initialized to infinity if not provided). The reason that mint is required is that it is generally set to a small greater-than-zero (but scene-specific) epsilon value, large enough so that a ray starting at the computed point of intersection between another ray and a surface doesn't incorrectly intersect the surface a very short distance along its extent. More details on this topic are in Section 3.1.4.

⟨*Ray Public Methods*⟩ ≡ 66
```
Ray() : mint(0.f), maxt(INFINITY), time(0.f), depth(0) { }
Ray(const Point &origin, const Vector &direction,
    float start, float end = INFINITY, float t = 0.f, int d = 0)
    : o(origin), d(direction), mint(start), maxt(end), time(t), depth(d) { }
```

For convenience we also provide another ray constructor that takes a "parent" ray; the idea is that when additional rays are being spawned from an intersection, it's useful to be able to copy the time value and set the depth value for the new ray based on the corresponding values from the previous ray.

⟨*Ray Public Methods*⟩ +≡ 66
```
Ray(const Point &origin, const Vector &direction, const Ray &parent,
    float start, float end = INFINITY)
    : o(origin), d(direction), mint(start), maxt(end),
      time(parent.time), depth(parent.depth+1) { }
```

Because a ray can be thought of as a function of a single parameter t, the Ray class overloads the function application operator for rays. This way, when we need to find the point at a particular position along a ray, we can write code like:

```
Ray r(Point(0,0,0), Vector(1,2,3));
Point p = r(1.7);
```

⟨*Ray Public Methods*⟩ +≡ 66
```
Point operator()(float t) const { return o + d * t; }
```

2.5.1 RAY DIFFERENTIALS

In order to be able to perform better antialiasing with the texture functions defined in Chapter 10, pbrt keeps track of some additional information with each ray that is traced. In Section 10.1, this information will be used to compute information so that the Texture class can estimate the projected area on the image plane of a small part of the scene. From this, the Texture can compute the texture's average value over that area, leading to a better final image.

RayDifferential is a subclass of Ray that contains additional information about two auxiliary rays. These extra rays represent camera rays offset one sample in the x and y direction from the main ray on the film plane. By determining the area that these three rays project to on an object being shaded, the Texture can estimate an area to average over for proper antialiasing.

INFINITY 1002
Point 63
Ray 66
RayDifferential 69
Texture 519
Vector 57

Because the RayDifferential class inherits from Ray, geometric interfaces in the system are written to take const Ray & parameters, so that either a Ray or RayDifferential can be passed to them. Only the routines related to antialiasing and texturing require RayDifferential parameters.

⟨*Geometry Declarations*⟩ +≡
```
class RayDifferential : public Ray {
public:
    ⟨RayDifferential Public Methods 69⟩
    ⟨RayDifferential Public Data 69⟩
};
```

The RayDifferential constructors mirror the Ray's constructors.

⟨*RayDifferential Public Methods*⟩ ≡ 69
```
RayDifferential() { hasDifferentials = false; }
RayDifferential(const Point &org, const Vector &dir, float start,
    float end = INFINITY, float t = 0.f, int d = 0)
        : Ray(org, dir, start, end, t, d) {
    hasDifferentials = false;
}
RayDifferential(const Point &org, const Vector &dir, const Ray &parent,
    float start, float end = INFINITY)
        : Ray(org, dir, start, end, parent.time, parent.depth+1) {
    hasDifferentials = false;
}
```

We provide a constructor to create RayDifferentials from Rays, but use the explicit keyword to prevent Rays from accidentally being converted to RayDifferentials. The constructor sets hasDifferentials to false initially because the neighboring rays, if any, are not known.

⟨*RayDifferential Public Methods*⟩ +≡ 69
```
explicit RayDifferential(const Ray &ray) : Ray(ray) {
    hasDifferentials = false;
}
```

⟨*RayDifferential Public Data*⟩ ≡ 69
```
bool hasDifferentials;
Point rxOrigin, ryOrigin;
Vector rxDirection, ryDirection;
```

Camera implementations in pbrt compute differentials for rays leaving the camera under the assumption that camera rays are spaced one pixel apart. The ScaleDifferentials() method below updates the differential rays for an actual sample spacing s.

Camera 302
INFINITY 1002
Point 63
Ray 66
Ray::depth 67
Ray::time 67
RayDifferential 69
RayDifferential::
hasDifferentials 69
Vector 57

⟨*RayDifferential Public Methods*⟩ +≡ 69
```
void ScaleDifferentials(float s) {
    rxOrigin = o + (rxOrigin - o) * s;
    ryOrigin = o + (ryOrigin - o) * s;
    rxDirection = d + (rxDirection - d) * s;
    ryDirection = d + (ryDirection - d) * s;
}
```

2.6 THREE-DIMENSIONAL BOUNDING BOXES

⟨*Geometry Declarations*⟩ +≡
```
class BBox {
public:
    ⟨BBox Public Methods 70⟩
    ⟨BBox Public Data 71⟩
};
```

The scenes that pbrt will render will often contain objects that are computationally expensive to process. For many operations, it is often useful to have a three-dimensional *bounding volume* that encloses an object. For example, if a ray does not pass through a particular bounding volume, pbrt can avoid processing all of the objects inside of it for that ray.

The measurable benefit of this technique is related to two factors: the expense of processing the bounding volume compared to the expense of processing the objects inside of it, and the tightness of the fit of the bounding box. If we have a very loose bound around an object, we will often incorrectly determine that its contents need to be examined further. However, in order to make the bounding volume a closer fit, it may be necessary to make the volume a complex object itself, thus increasing the expense of processing it.

There are many choices for bounding volumes; pbrt uses *axis-aligned bounding boxes* (AABBs). Other popular choices are spheres and *oriented bounding boxes* (OBBs). An AABB can be described by one of its vertices and three lengths, each representing the distance spanned along the x, y, and z coordinate axes. Alternatively, two opposite vertices of the box can describe it. We chose the two-point representation for pbrt's BBox class; it stores the positions of the vertex with minimum x, y, and z values and of the one with maximum x, y, and z. A 2D illustration of a bounding box and its representation is shown in Figure 2.8.

The default BBox constructor sets the extent to be degenerate; by violating the invariant that pMin.x <= pMax.x and similarly for the other dimensions, this convention ensures that any operations done with this box (e.g., Union()) will have the correct result for a completely empty box.

⟨*BBox Public Methods*⟩ ≡ 70
```
BBox() {
    pMin = Point( INFINITY,  INFINITY,  INFINITY);
    pMax = Point(-INFINITY, -INFINITY, -INFINITY);
}
```

BBox 70
BBox::pMax 71
BBox::pMin 71
INFINITY 1002
Point 63
Ray::d 67
Ray::o 67
RayDifferential::
rxDirection 69
RayDifferential::rxOrigin 69
RayDifferential::
ryDirection 69
RayDifferential::ryOrigin 69

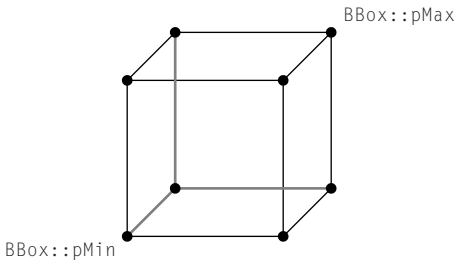


Figure 2.8: An Example Axis-Aligned Bounding Box. The BBox class stores only the coordinates of the minimum and maximum points of the box. All other box corners are implicit in this representation.

⟨*BBox Public Data*⟩ ≡ 70
```
Point pMin, pMax;
```

It is also useful to be able to initialize a BBox to enclose a single point:

⟨*BBox Public Methods*⟩ +≡ 70
```
BBox(const Point &p) : pMin(p), pMax(p) { }
```

If the caller passes two corner points (p1 and p2) to define the box, since p1 and p2 are not necessarily chosen so that p1.x <= p2.x, and so on, the constructor needs to find their component-wise minimum and maximum values.

⟨*BBox Public Methods*⟩ +≡ 70
```
BBox(const Point &p1, const Point &p2) {
    pMin = Point(min(p1.x, p2.x), min(p1.y, p2.y), min(p1.z, p2.z));
    pMax = Point(max(p1.x, p2.x), max(p1.y, p2.y), max(p1.z, p2.z));
}
```

Given a bounding box and a point, the BBox::Union() function computes and returns a new bounding box that encompasses that point as well as the space that the original box encompassed:

⟨*BBox Method Definitions*⟩ ≡
```
BBox Union(const BBox &b, const Point &p) {
    BBox ret = b;
    ret.pMin.x = min(b.pMin.x, p.x);
    ret.pMin.y = min(b.pMin.y, p.y);
    ret.pMin.z = min(b.pMin.z, p.z);
    ret.pMax.x = max(b.pMax.x, p.x);
    ret.pMax.y = max(b.pMax.y, p.y);
    ret.pMax.z = max(b.pMax.z, p.z);
    return ret;
}
```

BBox 70
BBox::pMax 71
BBox::pMin 71
BBox::Union() 71
Point 63

It is similarly possible to construct a new box that bounds the space encompassed by two other bounding boxes. The definition of this function is similar to the earlier `Union()` method that takes a `Point`; the difference is that the `pMin` and `pMax` of the second box are used for the `min()` and `max()` tests, respectively.

⟨*BBox Public Methods*⟩ +≡ 70
```
friend BBox Union(const BBox &b, const BBox &b2);
```

It is easy to determine if two `BBox`es overlap by seeing if their extents overlap in all of x, y, and z:

⟨*BBox Public Methods*⟩ +≡ 70
```
bool Overlaps(const BBox &b) const {
    bool x = (pMax.x >= b.pMin.x) && (pMin.x <= b.pMax.x);
    bool y = (pMax.y >= b.pMin.y) && (pMin.y <= b.pMax.y);
    bool z = (pMax.z >= b.pMin.z) && (pMin.z <= b.pMax.z);
    return (x && y && z);
}
```

Three simple 1D containment tests determine if a given point is inside the bounding box:

⟨*BBox Public Methods*⟩ +≡ 70
```
bool Inside(const Point &pt) const {
    return (pt.x >= pMin.x && pt.x <= pMax.x &&
            pt.y >= pMin.y && pt.y <= pMax.y &&
            pt.z >= pMin.z && pt.z <= pMax.z);
}
```

The `BBox::Expand()` method pads the bounding box by a constant factor.

⟨*BBox Public Methods*⟩ +≡ 70
```
void Expand(float delta) {
    pMin -= Vector(delta, delta, delta);
    pMax += Vector(delta, delta, delta);
}
```

Methods for computing the surface area of the six faces of the box and the volume inside of it are also frequently useful.

⟨*BBox Public Methods*⟩ +≡ 70
```
float SurfaceArea() const {
    Vector d = pMax - pMin;
    return 2.f * (d.x * d.y + d.x * d.z + d.y * d.z);
}
```

⟨*BBox Public Methods*⟩ +≡ 70
```
float Volume() const {
    Vector d = pMax - pMin;
    return d.x * d.y * d.z;
}
```

BBox 70
BBox::Expand() 72
BBox::pMax 71
BBox::pMin 71
Point 63
Vector 57

The BBox::MaximumExtent() method tells the caller which of the three axes is longest. This is useful, for example, when deciding along which axis to subdivide when building a kd-tree (see Section A.8).

⟨*BBox Public Methods*⟩ +≡ 70
```
int MaximumExtent() const {
    Vector diag = pMax - pMin;
    if (diag.x > diag.y && diag.x > diag.z)
        return 0;
    else if (diag.y > diag.z)
        return 1;
    else
        return 2;
}
```

In some cases, it's also useful to use array indexing to select between the two points at the corners of the box.

⟨*BBox Public Methods*⟩ +≡ 70
```
const Point &operator[](int i) const;
Point &operator[](int i);
```

The Lerp() method linearly interpolates between the corners of the box by the given amount, and Offset() returns the position of a point relative to the corners of the box, where a point at the minimum corner has offset (0, 0, 0), a point at the maximum corner has offset (1, 1, 1), and so forth.

⟨*BBox Public Methods*⟩ +≡ 70
```
Point Lerp(float tx, float ty, float tz) const {
    return Point(::Lerp(tx, pMin.x, pMax.x), ::Lerp(ty, pMin.y, pMax.y),
                 ::Lerp(tz, pMin.z, pMax.z));
}
```

⟨*BBox Public Methods*⟩ +≡ 70
```
Vector Offset(const Point &p) const {
    return Vector((p.x - pMin.x) / (pMax.x - pMin.x),
                  (p.y - pMin.y) / (pMax.y - pMin.y),
                  (p.z - pMin.z) / (pMax.z - pMin.z));
}
```

BBox 70
BBox::MaximumExtent() 73
BBox::pMax 71
BBox::pMin 71
Lerp() 1000
Point 63
Vector 57

Finally, the BBox provides a method that returns the center and radius of a sphere that bounds the bounding box. In general, this may give a far looser fit than a sphere that bounded the original contents of the BBox directly, although it is a useful method to have available. For example, in Chapter 14, we use this method to get a sphere that completely bounds the scene in order to generate a random ray that is likely to intersect the scene geometry.

⟨*BBox Method Definitions*⟩ +≡

```
void BBox::BoundingSphere(Point *c, float *rad) const {
    *c = .5f * pMin + .5f * pMax;
    *rad = Inside(*c) ? Distance(*c, pMax) : 0.f;
}
```

2.7 TRANSFORMATIONS

In general, a *transformation* $\mathbf{T}$ is a mapping from points to points and from vectors to vectors:

$$\mathrm{p}' = \mathbf{T}(\mathrm{p}) \qquad \mathbf{v}' = \mathbf{T}(\mathbf{v}).$$

The transformation $\mathbf{T}$ may be an arbitrary procedure. However, we will consider a subset of all possible transformations in this chapter. In particular, they will be

- *Linear:* If $\mathbf{T}$ is an arbitrary linear transformation and s is an arbitrary scalar, then $\mathbf{T}(s\mathbf{v}) = s\mathbf{T}(\mathbf{v})$ and $\mathbf{T}(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{T}(\mathbf{v}_1) + \mathbf{T}(\mathbf{v}_2)$. These two properties can greatly simplify reasoning about transformations.
- *Continuous:* Roughly speaking, $\mathbf{T}$ maps the neighborhoods around p and $\mathbf{v}$ to neighborhoods around p' and $\mathbf{v}'$.
- *One-to-one and invertible:* For each p, $\mathbf{T}$ maps p to a single unique p'. Furthermore, there exists an inverse transform $\mathbf{T}^{-1}$ that maps p' back to p.

We will often want to take a point, vector, or normal defined with respect to one coordinate frame and find its coordinate values with respect to another frame. Using basic properties of linear algebra, a 4×4 matrix can be shown to express the linear transformation of a point or vector from one frame to another. Furthermore, such a 4×4 matrix suffices to express all linear transformations of points and vectors within a fixed frame, such as translation in space or rotation around a point. Therefore, there are different (and incompatible!) ways that a matrix can be interpreted:

- *Transformation of the frame:* Given a point, the matrix could express how to compute a *new* point in the same frame that represents the transformation of the original point (e.g., by translating it in some direction).
- *Transformation from one frame to another:* A matrix can express the coordinates of a point or vector in a new frame in terms of the coordinates in the original frame.

Most uses of transformations in `pbrt` are for transforming points from one frame to another.

In general, transformations make it possible to work in the most convenient coordinate space. For example, we can write routines that define a virtual camera assuming that the camera is located at the origin, looks down the z axis, and has the y axis pointing up and the x axis pointing right. These assumptions greatly simplify the camera implementation. Then, to place the camera at any point in the scene looking in any direction, we just construct a transformation that maps points in the scene's coordinate system to the camera's coordinate system. (See Section 6.1.1 for more information about camera coordinate spaces in `pbrt`.)

BBox 70
BBox::Inside() 72
BBox::pMax 71
BBox::pMin 71
Distance() 65
Point 63

2.7.1 HOMOGENEOUS COORDINATES

Given a frame defined by $(\mathrm{p}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$, there is ambiguity between the representation of a point $(\mathrm{p}_x, \mathrm{p}_y, \mathrm{p}_z)$ and a vector $(\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z)$ with the same (x, y, z) coordinates. Using the representations of points and vectors introduced at the start of the chapter, we can write the point as the inner product $[s_1\, s_2\, s_3\, 1][\mathbf{v_1}\, \mathbf{v_2}\, \mathbf{v_3}\, \mathrm{p_o}]^T$ and the vector as the inner product $[s'_1\, s'_2\, s'_3\, 0][\mathbf{v_1}\, \mathbf{v_2}\, \mathbf{v_3}\, \mathrm{p_o}]^T$. These four vectors of three s_i values and a zero or one are called the *homogeneous* representations of the point and the vector. The fourth coordinate of the homogeneous representation is sometimes called the *weight*. For a point, its value can be any scalar other than zero: the homogeneous points $[1, 3, -2, 1]$ and $[-2, -6, 4, -2]$ describe the same Cartesian point $(1, 3, -2)$. In general, homogeneous points obey the identity

$$(x, y, z, w) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right).$$

We will use these facts to see how a transformation matrix can describe how points and vectors in one frame can be mapped to another frame. Consider a matrix $\mathbf{M}$ that describes the transformation from one coordinate system to another:

$$\mathbf{M} = \begin{pmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{pmatrix}.$$

(In this book, we define matrix element indices starting from zero, so that equations and source code correspond more directly.) Then if the transformation represented by $\mathbf{M}$ is applied to the x axis vector $(1, 0, 0)$, we have

$$\mathbf{Mx} = \mathbf{M}[1\ 0\ 0\ 0]^T = [m_{0,0}\, m_{1,0}\, m_{2,0}\, m_{3,0}]^T.$$

Thus, directly reading the columns of the matrix shows how the basis vectors and the origin of the current coordinate system are transformed by the matrix:

$$\mathbf{My} = [m_{0,1}\, m_{1,1}\, m_{2,1}\, m_{3,1}]^T$$
$$\mathbf{Mz} = [m_{0,2}\, m_{1,2}\, m_{2,2}\, m_{3,2}]^T$$
$$\mathbf{Mp} = [m_{0,3}\, m_{1,3}\, m_{2,3}\, m_{3,3}]^T.$$

In general, by characterizing how the basis is transformed, we know how any point or vector specified in terms of that basis is transformed. Because points and vectors in the current coordinate system are expressed in terms of the current coordinate system's frame, applying the transformation to them directly is equivalent to applying the transformation to the current coordinate system's basis and finding their coordinates in terms of the transformed basis.

We will not use homogeneous coordinates explicitly in our code; there is no `Homogeneous` class. However, the various transformation routines in the next section will implicitly convert points, vectors, and normals to homogeneous form, transform the homogeneous points, and then convert them back before returning the result. This isolates the details of homogeneous coordinates in one place (namely, the implementation of transformations).

⟨*Transform Declarations*⟩ ≡
```
class Transform {
public:
    ⟨Transform Public Methods 77⟩
private:
    ⟨Transform Private Data 76⟩
};
```

A transformation is represented by the elements of the matrix `m[4][4]`, a `Matrix4x4` object. The low-level `Matrix4x4` class is defined in Section A.6.2. `m` is stored in *row-major* form, so element `m[i][j]` corresponds to $m_{i,j}$, where i is the row number and j is the column number. For convenience, the `Transform` also stores the inverse of the matrix `m` in the `Transform::mInv` member; for `pbrt`'s needs, it is better to have the inverse easily available than to repeatedly compute it as needed.

This representation of transformations is relatively memory hungry: assuming 4 bytes of storage for a `float` value, a `Transform` requires 128 bytes of storage. Used naïvely, this approach can be wasteful; if a scene has millions of shapes but only a few thousand unique transformations, there's no reason to redundantly store the same transform many times in memory. Therefore, `Shapes` in `pbrt` store a pointer to a `Transform`, and the scene specification code defined in Section B.3.5 uses a `TransformCache` to ensure that all shapes that share the same transformation point to a single instance of that transformation in memory.

This decision to share transformations implies a loss of flexibility, however: the elements of a `Transform` shouldn't be modified after it is created if the `Transform` is shared by multiple objects in the scene (and those objects don't expect it to be changing.) This limitation isn't a problem in practice, since the transformations in a scene are typically created when `pbrt` parses the scene description file and don't need to change later at rendering time.

⟨*Transform Private Data*⟩ ≡ 76
```
Matrix4x4 m, mInv;
```

2.7.2 BASIC OPERATIONS

When a new `Transform` is created, it defaults to the *identity transformation*—the transformation that maps each point and each vector to itself. This transformation is represented by the *identity matrix*:

$$\mathrm{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The implementation here relies on the default `Matrix4x4` constructor to fill in the identity matrix for `m` and `mInv`.

`Matrix4x4` 1021
`Shape` 108
`Transform` 76
`Transform::mInv` 76
`TransformCache` 1066

⟨*Transform Public Methods*⟩ ≡ 76
```
Transform() { }
```

A `Transform` can also be created from a given matrix. In this case, the given matrix must be explicitly inverted.

⟨*Transform Public Methods*⟩ +≡ 76
```
Transform(const float mat[4][4]) {
    m = Matrix4x4(mat[0][0], mat[0][1], mat[0][2], mat[0][3],
                  mat[1][0], mat[1][1], mat[1][2], mat[1][3],
                  mat[2][0], mat[2][1], mat[2][2], mat[2][3],
                  mat[3][0], mat[3][1], mat[3][2], mat[3][3]);
    mInv = Inverse(m);
}
```

⟨*Transform Public Methods*⟩ +≡ 76
```
Transform(const Matrix4x4 &mat)
    : m(mat), mInv(Inverse(mat)) {
}
```

The most commonly used constructor takes a reference to the transformation matrix along with an explicitly provided inverse. This is a superior approach to computing the inverse in the constructor because many geometric transformations have very simple inverses and we can avoid the expense and potential loss of numeric precision from computing a general 4×4 matrix inverse. Of course, this places the burden on the caller to make sure that the supplied inverse is correct.

⟨*Transform Public Methods*⟩ +≡ 76
```
Transform(const Matrix4x4 &mat, const Matrix4x4 &minv)
   : m(mat), mInv(minv) {
}
```

A `Transform` representing the inverse of a `Transform` can be returned by just swapping the roles of `mInv` and `m`.

⟨*Transform Public Methods*⟩ +≡ 76
```
friend Transform Inverse(const Transform &t) {
    return Transform(t.mInv, t.m);
}
```

We provide `Transform` equality (and inequality) testing methods; their implementation is straightforward and not included here. `Transform` also provides an `IsIdentity()` method that checks to see if the transformation is the identity.

Inverse() 1021
Matrix4x4 1021
Transform 76
Transform::m 76
Transform::mInv 76

2.7.3 TRANSLATIONS

One of the simplest transformations is the *translation transformation*, $\mathrm{T}(\Delta x, \Delta y, \Delta z)$. When applied to a point p, it translates p's coordinates by Δx, Δy, and Δz, as shown in Figure 2.9. As an example, $\mathrm{T}(2, 2, 1)(x, y, z) = (x + 2, y + 2, z + 1)$.

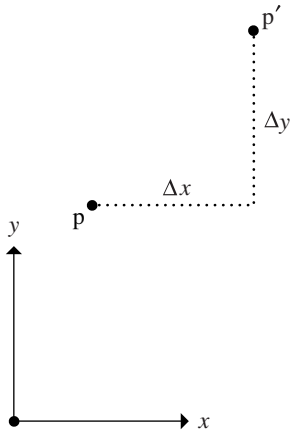


Figure 2.9: Translation in 2D. Adding offsets Δx and Δy to a point's coordinates correspondingly changes its position in space.

Translation has some simple properties:

$$\begin{aligned}
\mathbf{T}(0, 0, 0) &= \mathbf{I} \\
\mathbf{T}(x_1, y_1, z_1)\mathbf{T}(x_2, y_2, z_2) &= \mathbf{T}(x_1 + x_2, y_1 + y_2, z_1 + z_2) \\
\mathbf{T}(x_1, y_1, z_1)\mathbf{T}(x_2, y_2, z_2) &= \mathbf{T}(x_2, y_2, z_2)\mathbf{T}(x_1, y_1, z_1) \\
\mathbf{T}^{-1}(x, y, z) &= \mathbf{T}(-x, -y, -z).
\end{aligned}$$

Translation only affects points, leaving vectors unchanged. In matrix form, the translation transformation is

$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

When we consider the operation of a translation matrix on a point, we see the value of homogeneous coordinates. Consider the product of the matrix for $\mathbf{T}(\Delta x, \Delta y, \Delta z)$ with a point p in homogeneous coordinates $[x\ y\ z\ 1]$:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{pmatrix}.$$

As expected, we have computed a new point with its coordinates offset by $(\Delta x, \Delta y, \Delta z)$. However, if we apply $\mathbf{T}$ to a vector $\mathbf{v}$, we have

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}.$$

The result is the same vector **v**. This makes sense because vectors represent directions, so translation leaves them unchanged.

We will define a routine that creates a new `Transform` matrix to represent a given translation—it is a straightforward application of the translation matrix equation. This routine fully initializes the `Transform` that is returned, also initializing the matrix that represents the inverse of the translation.

⟨*Transform Method Definitions*⟩ ≡

```
Transform Translate(const Vector &delta) {
    Matrix4x4 m(1, 0, 0, delta.x,
                0, 1, 0, delta.y,
                0, 0, 1, delta.z,
                0, 0, 0,       1);
    Matrix4x4 minv(1, 0, 0, -delta.x,
                   0, 1, 0, -delta.y,
                   0, 0, 1, -delta.z,
                   0, 0, 0,        1);
    return Transform(m, minv);
}
```

2.7.4 SCALING

Another basic transformation is the *scale transformation*, $\mathrm{S}(s_x, s_y, s_z)$. It has the effect of taking a point or vector and multiplying its components by scale factors in x, y, and z: $\mathrm{S}(2, 2, 1)(x, y, z) = (2x, 2y, z)$. It has the following basic properties:

$$\begin{aligned} \mathrm{S}(1, 1, 1) &= \mathrm{I} \\ \mathrm{S}(x_1, y_1, z_1)\mathrm{S}(x_2, y_2, z_2) &= \mathrm{S}(x_1x_2, y_1y_2, z_1z_2) \\ \mathrm{S}^{-1}(x, y, z) &= \mathrm{S}\left(\frac{1}{x}, \frac{1}{y}, \frac{1}{z}\right). \end{aligned}$$

We can differentiate between *uniform scaling*, where all three scale factors have the same value, and *nonuniform scaling*, where they may have different values. The general scale matrix is

$$\mathrm{S}(x, y, z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Matrix4x4 1021
Transform 76
Vector 57

⟨*Transform Method Definitions*⟩ +≡
```
Transform Scale(float x, float y, float z) {
    Matrix4x4 m(x, 0, 0, 0,
                0, y, 0, 0,
                0, 0, z, 0,
                0, 0, 0, 1);
    Matrix4x4 minv(1.f/x,     0,     0,     0,
                      0,  1.f/y,     0,     0,
                      0,     0,  1.f/z,     0,
                      0,     0,     0,     1);
    return Transform(m, minv);
}
```

It's useful to be able to test if a transformation has a scaling term in it; an easy way to do this is to transform the three coordinate axes and see if any of their lengths are appreciably different from one.

⟨*Transform Public Methods*⟩ +≡ 76
```
bool HasScale() const {
    float la2 = (*this)(Vector(1,0,0)).LengthSquared();
    float lb2 = (*this)(Vector(0,1,0)).LengthSquared();
    float lc2 = (*this)(Vector(0,0,1)).LengthSquared();
#define NOT_ONE(x) ((x) < .999f || (x) > 1.001f)
    return (NOT_ONE(la2) || NOT_ONE(lb2) || NOT_ONE(lc2));
#undef NOT_ONE
}
```

2.7.5 x, y, AND z AXIS ROTATIONS

Another useful type of transformation is the *rotation transformation*, $\mathbf{R}$. In general, we can define an arbitrary axis from the origin in any direction and then rotate around that axis by a given angle. The most common rotations of this type are around the x, y, and z coordinate axes. We will write these rotations as $\mathbf{R}_x(\theta)$, $\mathbf{R}_y(\theta)$, and so on. The rotation around an arbitrary axis (x, y, z) is denoted by $\mathbf{R}_{(x,y,z)}(\theta)$.

Rotations also have some basic properties:

$$\begin{aligned}
\mathbf{R}_a(0) &= \mathbf{I} \\
\mathbf{R}_a(\theta_1)\mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_1 + \theta_2) \\
\mathbf{R}_a(\theta_1)\mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_2)\mathbf{R}_a(\theta_1) \\
\mathbf{R}_a^{-1}(\theta) &= \mathbf{R}_a(-\theta) = \mathbf{R}_a^T(\theta),
\end{aligned}$$

where $\mathbf{R}^T$ is the matrix transpose of $\mathbf{R}$. This last property, that the inverse of $\mathbf{R}$ is equal to its transpose, stems from the fact that $\mathbf{R}$ is an *orthogonal matrix*; its upper 3×3 components are all orthogonal to each other. Fortunately, the transpose is much easier to compute than a full matrix inverse.

Matrix4x4 1021
Transform 76
Vector 57
Vector::LengthSquared() 62

For a left-handed coordinate system, the matrix for rotation around the x axis is

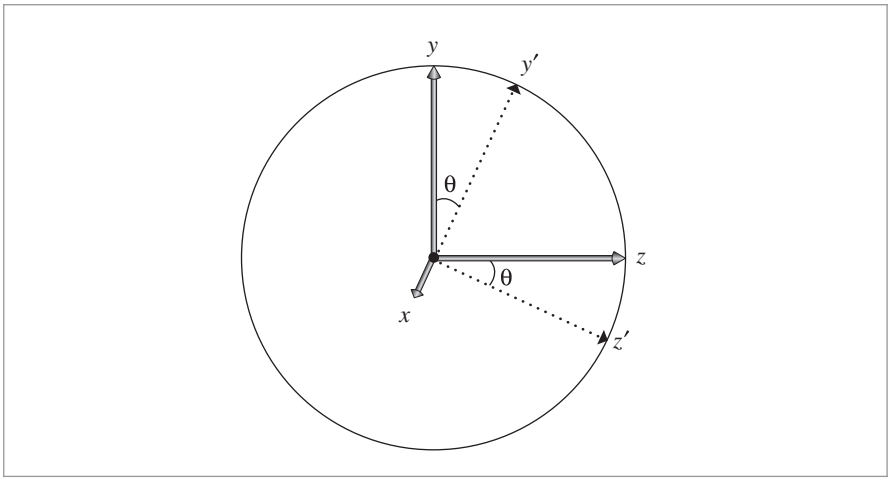


Figure 2.10: Rotation by an angle θ about the x axis leaves the x coordinate unchanged. The y and z axes are mapped to the vectors given by the dashed lines; y and z coordinates move accordingly.

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Figure 2.10 gives an intuition for how this matrix works. It's easy to see that it leaves the x axis unchanged:

$$\mathbf{R}_x(\theta)[1\,0\,0\,0]^T = [1\,0\,0\,0]^T.$$

It maps the y axis (0, 1, 0) to $(0, \cos\theta, \sin\theta)$ and the z axis to $(0, -\sin\theta, \cos\theta)$. The y and z axes remain in the same plane, perpendicular to the x axis, but are rotated by the given angle. An arbitrary point in space is similarly rotated about the x axis by this transformation while staying in the same yz plane as it was originally.

The implementation of the `RotateX()` function is straightforward.

⟨*Transform Method Definitions*⟩ +≡

```
Transform RotateX(float angle) {
    float sin_t = sinf(Radians(angle));
    float cos_t = cosf(Radians(angle));
    Matrix4x4 m(1,     0,      0, 0,
                0, cos_t, -sin_t, 0,
                0, sin_t,  cos_t, 0,
                0,     0,      0, 1);
    return Transform(m, Transpose(m));
}
```

Matrix4x4 1021
RotateX() 81
Transform 76
Transpose() 1021

Similarly, for rotation around y and z, we have

$$\mathrm{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathrm{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The implementations of `RotateY()` and `RotateZ()` follow directly and are not included here.

2.7.6 ROTATION AROUND AN ARBITRARY AXIS

We also provide a routine to compute the transformation that represents rotation around an arbitrary axis. The usual derivation of this matrix is based on computing rotations that map the given axis to a fixed axis (e.g., z), performing the rotation there, and then rotating the fixed axis back to the original axis. A more elegant derivation can be constructed with vector algebra.

Consider a normalized direction vector $\mathbf{a}$ that gives the axis to rotate around by angle θ, and a vector $\mathbf{v}$ to be rotated (Figure 2.11). First, we can compute the vector $\mathbf{v_c}$ along the axis $\mathbf{a}$ that is in the plane through the end point of $\mathbf{v}$ and is parallel to $\mathbf{a}$. Assuming $\mathbf{v}$ and $\mathbf{a}$ form an angle α, we have

$$\mathbf{v_c} = \mathbf{a}\,\|\mathbf{v}\|\cos\alpha = \mathbf{a}(\mathbf{v}\cdot\mathbf{a}).$$

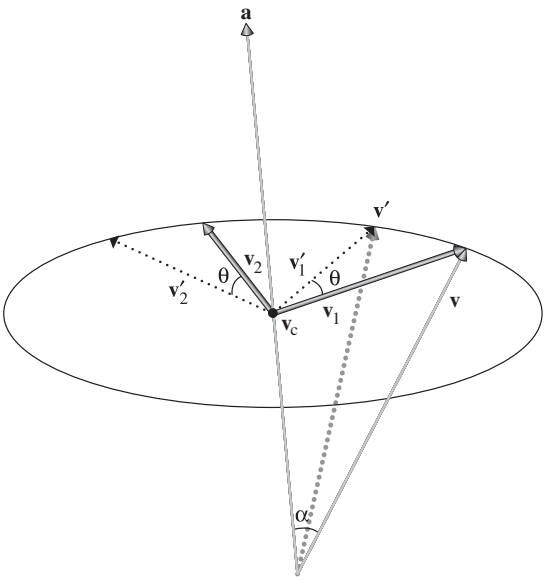


Figure 2.11: A vector $\mathbf{v}$ can be rotated around an arbitrary axis $\mathbf{a}$ by constructing a coordinate system $(\mathrm{p}, \mathbf{v}_1, \mathbf{v}_2)$ in the plane perpendicular to the axis that passes through $\mathbf{v}$'s end point and rotating the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ about p. Applying this rotation to the axes of the coordinate system $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ gives the general rotation matrix for this rotation.

We now compute a pair of basis vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ in this plane. Trivially, one of them is

$$\mathbf{v}_1 = \mathbf{v} - \mathbf{v}_c,$$

and the other can be computed with a cross product

$$\mathbf{v}_2 = (\mathbf{v}_1 \times \mathbf{a}).$$

Because $\mathbf{a}$ is normalized, $\mathbf{v}_1$ and $\mathbf{v}_2$ have the same length, equal to the length of the vector between $\mathbf{v}$ and $\mathbf{v}_c$. To now compute the rotation by an angle θ about $\mathbf{v}_c$ in the plane of rotation, the rotation formulas earlier give us

$$\mathbf{v}' = \mathbf{v}_c + \mathbf{v}_1 \cos\theta + \mathbf{v}_2 \sin\theta.$$

To convert this to a rotation matrix, we apply this formula to the basis vectors (1, 0, 0), (0, 1, 0), and (0, 0, 1) to get the values of the rows of the matrix. The result of all this is encapsulated in the following function. As with the other rotation matrices, the inverse is equal to the transpose.

⟨*Transform Method Definitions*⟩ +≡

```
Transform Rotate(float angle, const Vector &axis) {
    Vector a = Normalize(axis);
    float s = sinf(Radians(angle));
    float c = cosf(Radians(angle));
    float m[4][4];

    m[0][0] = a.x * a.x + (1.f - a.x * a.x) * c;
    m[0][1] = a.x * a.y * (1.f - c) - a.z * s;
    m[0][2] = a.x * a.z * (1.f - c) + a.y * s;
    m[0][3] = 0;

    m[1][0] = a.x * a.y * (1.f - c) + a.z * s;
    m[1][1] = a.y * a.y + (1.f - a.y * a.y) * c;
    m[1][2] = a.y * a.z * (1.f - c) - a.x * s;
    m[1][3] = 0;

    m[2][0] = a.x * a.z * (1.f - c) - a.y * s;
    m[2][1] = a.y * a.z * (1.f - c) + a.x * s;
    m[2][2] = a.z * a.z + (1.f - a.z * a.z) * c;
    m[2][3] = 0;

    m[3][0] = 0;
    m[3][1] = 0;
    m[3][2] = 0;
    m[3][3] = 1;

    Matrix4x4 mat(m);
    return Transform(mat, Transpose(mat));
}
```

Matrix4x4 1021
Transform 76
Transpose() 1021
Vector 57
Vector::Normalize() 63

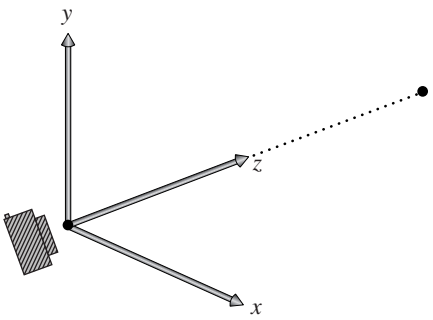


Figure 2.12: Given a camera position, the position being looked at from the camera, and an "up" direction, the look-at transformation describes a transformation from a viewing coordinate system where the camera is at the origin looking down the $+z$ axis, and the $+y$ axis is along the up direction.

2.7.7 THE LOOK-AT TRANSFORMATION

The *look-at transformation* is particularly useful for placing a camera in the scene. The caller specifies the desired position of the camera, a point the camera is looking at, and an "up" vector that orients the camera along the viewing direction implied by the first two parameters. All of these values are given in world space coordinates. The look-at construction then gives a transformation between camera space and world space (Figure 2.12).

In order to find the entries of the look-at transformation matrix, we use principles described earlier in this section: the columns of a transformation matrix give the effect of the transformation on the basis of a coordinate system.

⟨*Transform Method Definitions*⟩ +≡
```
Transform LookAt(const Point &pos, const Point &look, const Vector &up) {
    float m[4][4];
    ⟨Initialize fourth column of viewing matrix 84⟩
    ⟨Initialize first three columns of viewing matrix 85⟩
    Matrix4x4 camToWorld(m);
    return Transform(Inverse(camToWorld), camToWorld);
}
```

The easiest column is the fourth one, which gives the point that the camera space origin, $[0\ 0\ 0\ 1]^T$, maps to in world space. This is clearly just the camera position, supplied by the user.

⟨*Initialize fourth column of viewing matrix*⟩ ≡ 84
```
m[0][3] = pos.x;
m[1][3] = pos.y;
m[2][3] = pos.z;
m[3][3] = 1;
```

Inverse() 1021
Matrix4x4 1021
Point 63
Transform 76
Vector 57

The other three columns aren't much more difficult. First, `LookAt()` computes the normalized direction vector from the camera location to the look-at point; this gives the vector coordinates that the z axis should map to and, thus, the third column of the matrix. (Camera space is defined with the viewing direction down the $+z$ axis.) The first column, giving the world space direction that the $+x$ axis in camera space maps to, is found by taking the cross product of the user-supplied "up" vector with the recently computed viewing direction vector. Finally, the "up" vector is recomputed by taking the cross product of the viewing direction vector with the transformed x axis vector, thus ensuring that the y and z axes are perpendicular and we have an orthonormal viewing coordinate system.

⟨*Initialize first three columns of viewing matrix*⟩ ≡ 84

```
Vector dir = Normalize(look - pos);
Vector left = Normalize(Cross(Normalize(up), dir));
Vector newUp = Cross(dir, left);
m[0][0] = left.x;
m[1][0] = left.y;
m[2][0] = left.z;
m[3][0] = 0.;
m[0][1] = newUp.x;
m[1][1] = newUp.y;
m[2][1] = newUp.z;
m[3][1] = 0.;
m[0][2] = dir.x;
m[1][2] = dir.y;
m[2][2] = dir.z;
m[3][2] = 0.;
```

2.8 APPLYING TRANSFORMATIONS

We can now define routines that perform the appropriate matrix multiplications to transform points and vectors. We will overload the function application operator to describe these transformations; this lets us write code like:

```
Point P = ...;
Transform T = ...;
Point newP = T(P);
```

2.8.1 POINTS

The point transformation routine takes a point (x, y, z) and implicitly represents it as the homogeneous column vector $[x\ y\ z\ 1]^T$. It then transforms the point by premultiplying this vector with the transformation matrix. Finally, it divides by w to convert back to a nonhomogeneous point representation. For efficiency, this method skips the divide by the homogeneous weight, w, when $w = 1$, which is common for most of the transformations that will be used in `pbrt`—only the projective transformations defined in Chapter 6 will require this divide.

`Cross()` 62
`LookAt()` 84
`Vector` 57
`Vector::Normalize()` 63

⟨*Transform Inline Functions*⟩ ≡
```
inline Point Transform::operator()(const Point &pt) const {
    float x = pt.x, y = pt.y, z = pt.z;
    float xp = m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z + m.m[0][3];
    float yp = m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z + m.m[1][3];
    float zp = m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z + m.m[2][3];
    float wp = m.m[3][0]*x + m.m[3][1]*y + m.m[3][2]*z + m.m[3][3];
    if (wp == 1.) return Point(xp, yp, zp);
    else          return Point(xp, yp, zp)/wp;
}
```

We also provide transformation methods that let the caller pass a pointer to an object for the result. This saves the expense of returning structures by value on the stack. Note that we copy the original (x, y, z) coordinates to local variables in case the result pointer points to the same location as `pt`. This way, these routines can be used even if a point is being transformed in place.

⟨*Transform Inline Functions*⟩ +≡
```
inline void Transform::operator()(const Point &pt,
                                  Point *ptrans) const {
    float x = pt.x, y = pt.y, z = pt.z;
    ptrans->x = m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z + m.m[0][3];
    ptrans->y = m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z + m.m[1][3];
    ptrans->z = m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z + m.m[2][3];
    float w   = m.m[3][0]*x + m.m[3][1]*y + m.m[3][2]*z + m.m[3][3];
    if (w != 1.) *ptrans /= w;
}
```

2.8.2 VECTORS

The transformations of vectors can be computed in a similar fashion. However, the multiplication of the matrix and the row vector is simplified since the implicit homogeneous w coordinate is zero.

⟨*Transform Inline Functions*⟩ +≡
```
inline Vector Transform::operator()(const Vector &v) const {
  float x = v.x, y = v.y, z = v.z;
  return Vector(m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z,
                m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z,
                m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z);
}
```

There is also a method allowing the caller to pass a pointer to the result vector. The code to do this has a similar design to the `Point` transformation code and is not shown here. This code will also be omitted for subsequent transformation methods.

Point 63
Transform 76
Vector 57

2.8.3 NORMALS

Normals do not transform in the same way that vectors do, as shown in Figure 2.13. Although tangent vectors transform in the straightforward way, normals require special

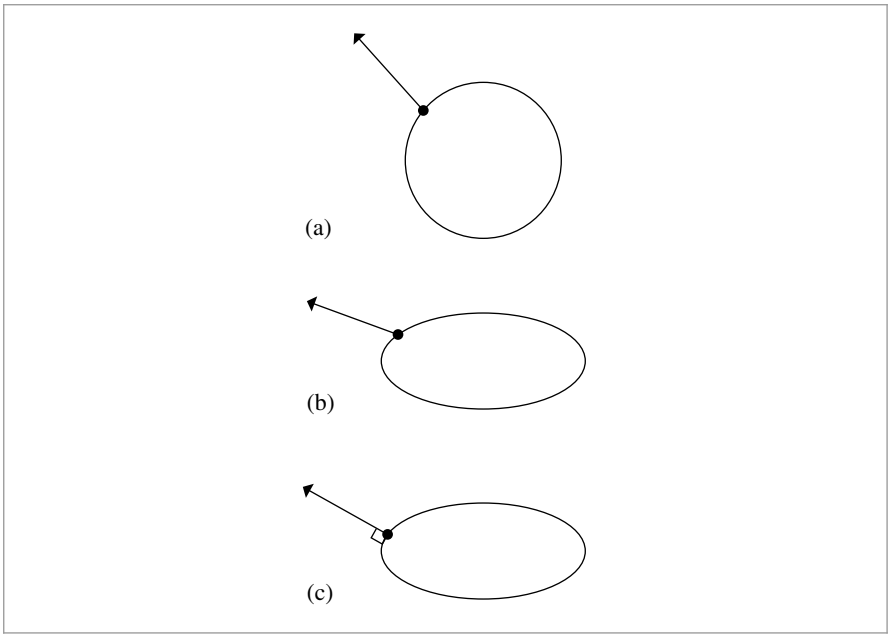


Figure 2.13: Transforming Surface Normals. (a) Original circle. (b) When scaling the circle to be half as tall in the y direction, simply treating the normal as a direction and scaling it in the same manner will lead to incorrect results. (c) A properly transformed normal.

treatment. Because the normal vector $\mathbf{n}$ and any tangent vector $\mathbf{t}$ on the surface are orthogonal by construction, we know that

$$\mathbf{n} \cdot \mathbf{t} = \mathbf{n}^T \mathbf{t} = 0.$$

When we transform a point on the surface by some matrix $\mathbf{M}$, the new tangent vector $\mathbf{t}'$ at the transformed point is $\mathbf{Mt}$. The transformed normal $\mathbf{n}'$ should be equal to $\mathbf{Sn}$ for some 4×4 matrix $\mathbf{S}$. To maintain the orthogonality requirement, we must have

$$\begin{aligned} 0 &= (\mathbf{n}')^T \mathbf{t}' \\ &= (\mathbf{Sn})^T \mathbf{Mt} \\ &= (\mathbf{n})^T \mathbf{S}^T \mathbf{Mt}. \end{aligned}$$

This condition holds if $\mathbf{S}^T\mathbf{M} = \mathbf{I}$, the identity matrix. Therefore, $\mathbf{S}^T = \mathbf{M}^{-1}$, and so $\mathbf{S} = \mathbf{M}^{-1^T}$, and we see that normals must be transformed by the inverse transpose of the transformation matrix. This detail is one of the main reasons why `Transforms` maintain their inverses.

`Transform` 76

`Vector` 57

Note that this method does not explicitly compute the transpose of the inverse when transforming normals. It just indexes into the inverse matrix in a different order (compare to the code for transforming `Vectors`).

⟨*Transform Inline Functions*⟩ +≡
```
inline Normal Transform::operator()(const Normal &n) const {
    float x = n.x, y = n.y, z = n.z;
    return Normal(mInv.m[0][0]*x + mInv.m[1][0]*y + mInv.m[2][0]*z,
                  mInv.m[0][1]*x + mInv.m[1][1]*y + mInv.m[2][1]*z,
                  mInv.m[0][2]*x + mInv.m[1][2]*y + mInv.m[2][2]*z);
}
```

2.8.4 RAYS

Transforming rays is straightforward: the method to do this transforms the constituent origin and direction and copies the other data members. `pbrt` also provides similar methods for transforming `RayDifferentials`.

⟨*Transform Inline Functions*⟩ +≡
```
inline Ray Transform::operator()(const Ray &r) const {
    Ray ret = r;
    (*this)(ret.o, &ret.o);
    (*this)(ret.d, &ret.d);
    return ret;
}
```

2.8.5 BOUNDING BOXES

The easiest way to transform an axis-aligned bounding box is to transform all eight of its corner vertices and then compute a new bounding box that encompasses those points. We will present code for this method below; one of the exercises for this chapter is to implement a technique to do this computation more efficiently.

⟨*Transform Method Definitions*⟩ +≡
```
BBox Transform::operator()(const BBox &b) const {
    const Transform &M = *this;
    BBox ret(        M(Point(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMax.z)));
    return ret;
}
```

BBox 70
Normal 65
Point 63
Ray 66
RayDifferential 69
Transform 76

2.8.6 COMPOSITION OF TRANSFORMATIONS

Having defined how the matrices representing individual types of transformations are constructed, we can now consider an aggregate transformation resulting from a series of individual transformations. Finally, we will see the real value of representing transformations with matrices.

Consider a series of transformations **ABC**. We'd like to compute a new transformation **T** such that applying **T** gives the same result as applying each of **A**, **B**, and **C** in reverse order; that is, $\mathbf{A}(\mathbf{B}(\mathbf{C}(\mathrm{p}))) = \mathbf{T}(\mathrm{p})$. Such a transformation **T** can be computed by multiplying the matrices of the transformations **A**, **B**, and **C** together. In `pbrt`, we can write:

```
Transform T = A * B * C;
```

Then we can apply `T` to `Points p` as usual, `Point pp = T(p)`, instead of applying each transformation in turn: `Point pp = A(B(C(p)))`.

We use the C++ `*` operator to compute the new transformation that results from post-multiplying a transformation with another transformation `t2`. In matrix multiplication, the (i, j)th element of the resulting matrix `ret` is the inner product of the ith row of the first matrix with the jth column of the second.

The inverse of the resulting transformation is equal to the product of `t2.mInv * mInv`. This is a result of the matrix identity

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}.$$

⟨*Transform Method Definitions*⟩ +≡

```
Transform Transform::operator*(const Transform &t2) const {
    Matrix4x4 m1 = Matrix4x4::Mul(m, t2.m);
    Matrix4x4 m2 = Matrix4x4::Mul(t2.mInv, mInv);
    return Transform(m1, m2);
}
```

2.8.7 TRANSFORMATIONS AND COORDINATE SYSTEM HANDEDNESS

Certain types of transformations change a left-handed coordinate system into a right-handed one, or vice versa. Some routines will need to know if the handedness of the source coordinate system is different from that of the destination. In particular, routines that want to ensure that a surface normal always points "outside" of a surface might need to flip the normal's direction after transformation if the handedness changes.

Fortunately, it is easy to tell if handedness is changed by a transformation: it happens only when the determinant of the transformation's upper-left 3×3 submatrix is negative.

⟨*Transform Method Definitions*⟩ +≡

```
bool Transform::SwapsHandedness() const {
    float det = ((m.m[0][0] *
                  (m.m[1][1] * m.m[2][2] -
                   m.m[1][2] * m.m[2][1])) -
                 (m.m[0][1] *
                  (m.m[1][0] * m.m[2][2] -
                   m.m[1][2] * m.m[2][0])) +
                 (m.m[0][2] *
                  (m.m[1][0] * m.m[2][1] -
                   m.m[1][1] * m.m[2][0])));
    return det < 0.f;
}
```

Matrix4x4 1021
Matrix4x4::Mul() 1021
Point 63
Transform 76

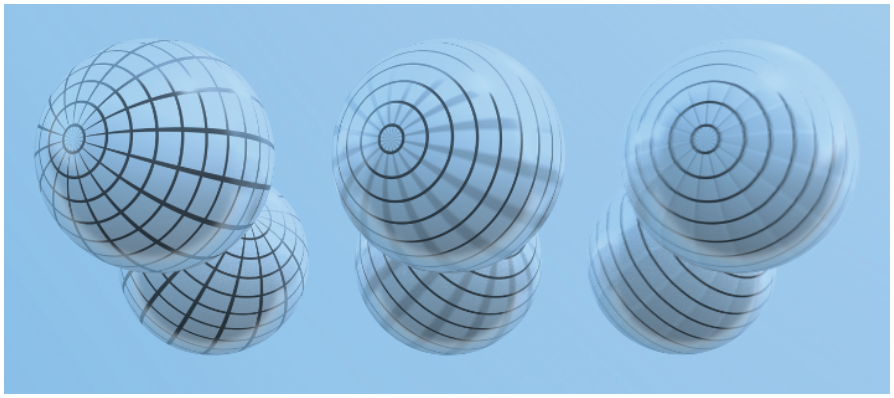


Figure 2.14: Spinning Spheres. Three spheres, spinning at different rates using the transformation animation code implemented in this section. Note that the reflections of the spheres are blurry as well as the spheres themselves.

* 2.9 ANIMATING TRANSFORMATIONS

`pbrt` supports *keyframe matrix animation* for cameras and geometric primitives in the scene. Rather than just supplying a single transformation to place the corresponding object in the scene, the user may supply a number of *keyframe* transformations, each one associated with a particular point in time. This makes it possible for the camera to move, and for objects in the scene to be moving during the time the simulated camera's shutter is open. Figure 2.14 shows three spheres animated using keyframe matrix animation in `pbrt`.

In general, the problem of interpolating between keyframe matrices is under-defined. As one example, if we have a rotation about the x axis of 179 degrees followed by another of 181 degrees, does this represent a small rotation of 2 degrees, or a large rotation of -358 degrees? For another example, consider two matrices where one is the identity and the other is a 180-degree rotation about the z axis. There are an infinite number of ways to go from one orientation to the other.

Keyframe matrix interpolation is an important problem in computer animation, where a number of different approaches have been developed. Fortunately, the problem of matrix interpolation in a renderer is generally less challenging than it is for animation systems for two reasons.

First, in a renderer like `pbrt`, we generally have a keyframe matrix at the camera shutter open time and another at the shutter close time; we only need to interpolate between the two of them across the time of a single image. In animation systems, the matrices are generally available at a lower time frequency, so that there are many frames between

* This section covers advanced topics and may be skipped on a first reading.

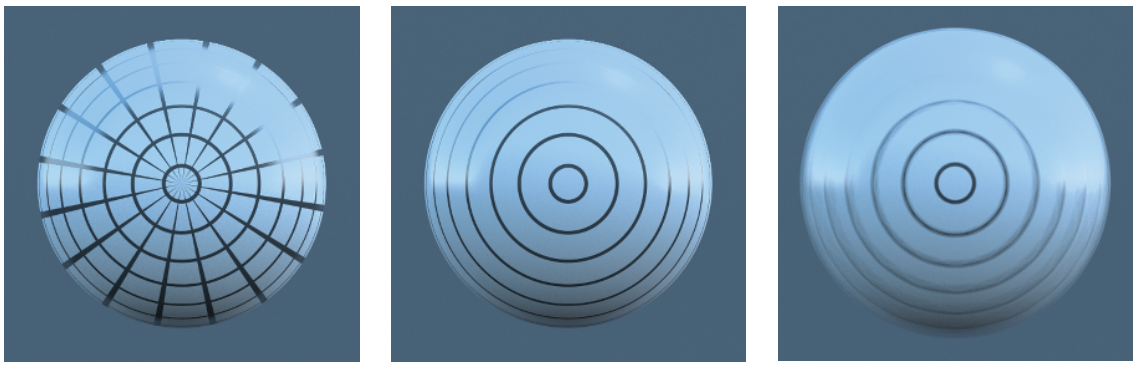


Figure 2.15: (a) Sphere with a grid of lines as a texture, not rotating. (b) Sphere rotating 90 degrees during the course of the frame, using the technique for interpolating transformations implemented in this section. (c) Sphere rotating 90 degrees using direct interpolation of matrix components to interpolate transformations. In this case, the animated sphere incorrectly grows larger. Furthermore, the lines toward the outside of the sphere, which should remain sharp, incorrectly become blurry.

pairs of keyframe matrices; as such, there's more opportunity to notice shortcomings in the interpolation.

Second, in a physically based renderer, the longer the period of time over which we need to interpolate the pair of matrices, the longer the virtual camera shutter is open and the more motion blur there will be in the final image; in general, the increased amount of motion blur will hide sins of the interpolation.

Here we will implement a method that interpolates between the transformations defined by the keyframe matrices. The most straightforward approach—directly interpolating the individual components of the two matrices—is not a good one, as it will generally lead to unexpected and undesirable results. For example, if the transformations apply different rotations, then even if we have a rigid-body motion, the intermediate matrices may scale the object, which is clearly undesirable. (If they have a full 180-degree rotation between them, the object may be scaled down to nothing at the middle of the interpolation!)

Figure 2.15 shows a sphere that rotates 90 degrees over the course of the frame; direct interpolation of matrix elements (on the right) gives a less accurate result than the approach implemented in this section (in the middle).

The approach used for transformation interpolation in `pbrt` is based on *matrix decomposition*—given an arbitrary transformation matrix $\mathbf{M}$, we decompose it into a concatenation of scale ($\mathbf{S}$), rotation ($\mathbf{R}$), and translation ($\mathbf{T}$) transformations,

$$\mathbf{M} = \mathbf{SRT},$$

where each of those components is independently interpolated and then the composite interpolated matrix is found by multiplying the three interpolated matrices together.

Interpolation of translation and scale can be performed easily and accurately with linear interpolation of the components of their matrices; interpolating rotations is more difficult. Before describing the matrix decomposition implementation in `pbrt`, we will

first introduce *quaternions*, an elegant representation of rotations that leads to elegant methods for interpolating rotations.

2.9.1 QUATERNIONS

Quaternions were originally invented by Sir William Rowan Hamilton in 1843 as a generalization of imaginary numbers. He determined that just as in two dimensions (x, y), where imaginary numbers could be defined as $x + y\mathrm{i}$, with $\mathrm{i}^2 = -1$, a generalization could be made to four dimensions, giving quaternions.

A quaternion is a four-tuple,

$$\mathbf{q} = (x, y, z, w) = w + x\mathrm{i} + y\mathrm{j} + z\mathrm{k}, \tag{2.4}$$

where i, j, and k are defined so that $\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1$.[2] Other important relationships between the components are that $\mathrm{ij} = \mathrm{k}$, and $\mathrm{ji} = -\mathrm{k}$. (Note that quaternion multiplication is not commutative.)

A quaternion can be represented as a quadruple $\mathbf{q} = (\mathbf{q}_x, \mathbf{q}_y, \mathbf{q}_z, \mathbf{q}_w)$, or as $\mathbf{q} = (\mathbf{q}_{xyz}, \mathbf{q}_w)$, where $\mathbf{q}_{xyz}$ is a 3-vector. We will use both representations interchangeably in this section.

Quaternion multiplication is defined by expanding out the product of two quaternions:

$$\mathbf{q}\mathbf{q}' = (\mathbf{q}_w + \mathbf{q}_x\mathrm{i} + \mathbf{q}_y\mathrm{j} + \mathbf{q}_z\mathrm{k})(\mathbf{q}'_w + \mathbf{q}'_x\mathrm{i} + \mathbf{q}'_y\mathrm{j} + \mathbf{q}'_z\mathrm{k}).$$

Collecting terms and using identities among the components like those listed above (e.g., $\mathrm{i}^2 = -1$), the result can be expressed concisely using vector cross and dot products:

$$\begin{aligned}(\mathbf{q}\mathbf{q}')_{xyz} &= \mathbf{q}_{xyz} \times \mathbf{q}'_{xyz} + \mathbf{q}_w\mathbf{q}'_{xyz} + \mathbf{q}'_w\mathbf{q}_{xyz} \\ (\mathbf{q}\mathbf{q}')_w &= \mathbf{q}_w\mathbf{q}'_w - (\mathbf{q}_{xyz} \cdot \mathbf{q}'_{xyz}).\end{aligned} \tag{2.5}$$

A unit quaternion (where $x^2 + y^2 + z^2 + w^2 = 1$) represents a rotation $(\mathbf{q}_{xyz} \sin\theta, \cos\theta)$ of angle 2θ around the unit axis $\hat{\mathbf{q}}_{xyz}$. Given a point p in homogeneous coordinates, the result of the rotation of the point by the quaternion is given by the quaternion product

$$\mathrm{p}' = \mathbf{q}\mathrm{p}\mathbf{q}^{-1}.$$

The implementation of the `Quaternion` class in `pbrt` is in the files `core/quaternion.h` and `core/quaternion.cpp`. The default constructor initializes a unit quaternion.

⟨*Quaternion Public Methods*⟩ ≡
```
Quaternion() { v = Vector(0., 0., 0.); w = 1.f; }
```

We use a `Vector` to represent the xyz components of the quaternion; this lets us take advantage of the already implemented methods of `Vector` in the implementation of some of the methods below.

Quaternion 92
Quaternion::v 93
Quaternion::w 93
Vector 57

2 Hamilton found the discovery of this relationship among the components compelling enough that he used a knife to carve the formula on the bridge he was crossing when it came to him.

⟨*Quaternion Public Data*⟩ ≡
```
Vector v;
float w;
```

Addition and subtraction of quaternions is performed component-wise. This follows directly from the definition in Equation (2.4). For example,

$$\begin{aligned}\mathbf{q} + \mathbf{q}' &= w + x\mathrm{i} + y\mathrm{j} + z\mathrm{k} + w' + x'\mathrm{i} + y'\mathrm{j} + z'\mathrm{k} \\ &= (w + w') + (x + x')\mathrm{i} + (y + y')\mathrm{j} + (z + z')\mathrm{k}.\end{aligned}$$

Other arithmetic methods (subtraction, multiplication, and division by a scalar) are defined and implemented similarly and won't be included here.

⟨*Quaternion Public Methods*⟩ +≡
```
Quaternion &operator+=(const Quaternion &q) {
    v += q.v;
    w += q.w;
    return *this;
}
```

The inner product of two quaternions is implemented by its `Dot()` method, and a quaternion can be normalized by dividing by its length.

⟨*Quaternion Inline Functions*⟩ ≡
```
inline float Dot(const Quaternion &q1, const Quaternion &q2) {
    return Dot(q1.v, q2.v) + q1.w * q2.w;
}
```

⟨*Quaternion Inline Functions*⟩ +≡
```
inline Quaternion Normalize(const Quaternion &q) {
    return q / sqrtf(Dot(q, q));
}
```

It's useful to be able to compute the transformation matrix that represents the same rotation as a quaternion. In particular, after interpolating rotations with quaternions in the `AnimatedTransform` class, we'll need to convert the interpolated rotation back to a transformation matrix to compute the final composite interpolated transformation.

To derive the rotation matrix for a quaternion, recall that the transformation of a point by a quaternion is given by $\mathrm{p}' = \mathbf{q}\mathrm{p}\mathbf{q}^{-1}$. We want a matrix $\mathbf{M}$ that performs the same transformation, so that $\mathrm{p}' = \mathbf{M}\mathrm{p}$. If we expand out the quaternion multiplication $\mathbf{q}\mathrm{p}\mathbf{q}^{-1}$ using Equation (2.5), simplify with the quaternion basis identities, collect terms, and represent the result in a matrix, we can determine that the following 3×3 matrix represents the same transformation:

$$\mathbf{M} = \begin{pmatrix} 1 - 2(\mathbf{q}_y^2 + \mathbf{q}_z^2) & 2(\mathbf{q}_x\mathbf{q}_y + \mathbf{q}_z\mathbf{q}_w) & 2\mathbf{q}_x\mathbf{q}_z - \mathbf{q}_y\mathbf{q}_w) \\ 2(\mathbf{q}_x\mathbf{q}_y - \mathbf{q}_z\mathbf{q}_w) & 1 - 2(\mathbf{q}_x^2 + \mathbf{q}_z^2) & 2(\mathbf{q}_y\mathbf{q}_z + \mathbf{q}_x\mathbf{q}_w) \\ 2(\mathbf{q}_x\mathbf{q}_z + \mathbf{q}_y\mathbf{q}_w) & 2(\mathbf{q}_y\mathbf{q}_z - \mathbf{q}_x\mathbf{q}_w) & 1 - 2(\mathbf{q}_x^2 + \mathbf{q}_y^2) \end{pmatrix}. \quad [2.6]$$

This computation is implemented in the method `Quaternion::ToTransform()`. We won't include its implementation here since it's a direct implementation of Equation (2.6).

AnimatedTransform 96
Dot() 60
Quaternion 92
Quaternion::Dot() 93
Quaternion::v 93
Quaternion::w 93
Vector 57

⟨*Quaternion Public Methods*⟩ +≡
```
Transform ToTransform() const;
```

Note that we could alternatively use the fact that a unit quaternion represents a rotation $(\mathbf{q}_{xyz} \sin\theta, \cos\theta)$ of angle 2θ around the unit axis $\hat{\mathbf{q}}_{xyz}$ to compute a rotation matrix. First we would compute the angle of rotation θ as $\theta = 2 \arccos \mathbf{q}_w$ and then we'd use the previously defined `Rotate()` function, passing it the axis $\hat{\mathbf{q}}_{xyz}$ and the rotation angle θ. However, this alternative would be substantially less efficient, requiring multiple calls to trigonometric functions, while the approach here only uses floating-point addition, subtraction, and multiplication.

It is also useful to be able to create a quaternion from a rotation matrix. For this purpose, `Quaternion` provides a constructor that takes a `Transform`. The appropriate quaternion can be computed by making use of relationships between elements of the rotation matrix in Equation (2.6) and quaternion components. For example, if we subtract the transpose of this matrix from itself, then the (0, 1) component of the resulting matrix has the value $-4\mathbf{q}_w\mathbf{q}_z$. Thus, given a particular instance of a rotation matrix with known values, it's possible to use a number of relationships like this between the matrix values and the quaternion components to generate a system of equations that can be solved for the quaternion components.

We won't include the details of the derivation or the actual implementation here in the text; for more information about how to derive this technique, including handling numerical robustness, see Shoemake (1991).

⟨*Quaternion Public Methods*⟩ +≡
```
Quaternion(const Transform &t);
```

2.9.2 QUATERNION INTERPOLATION

The last quaternion function we will define, `Slerp()`, interpolates between two quaternions using spherical linear interpolation. Spherical linear interpolation gives constant speed motion along great circle arcs on the surface of a sphere and consequently has two desirable properties for interpolating rotations:

- The interpolated rotation path exhibits *torque minimization*: the path to get between two rotations is the shortest possible path in rotation space.
- The interpolation has *constant angular velocity*: the relationship between change in the animation parameter t and the change in the resulting rotation is constant over the course of interpolation (in other words, the speed of interpolation is constant across the interpolation range).

See the "Further Reading" section at the end of the chapter for references that discuss more rigorously what characteristics a good interpolated rotation should have.

Spherical linear interpolation for quaternions was originally presented by Shoemake (1985) as follows, where two quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$ are given and $t \in [0, 1]$ is the parameter value to interpolate between them:

Quaternion 92
Rotate() 83
Slerp() 96
Transform 76

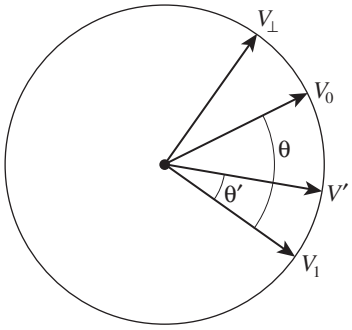


Figure 2.16: To understand quaternion spherical linear interpolation, consider in 2D two vectors on the unit sphere, $\mathbf{v_0}$ and $\mathbf{v_1}$, with angle θ between them. We'd like to be able to compute the interpolated vector at some angle θ' between the two of them. To do so, we can find a vector that's orthogonal to $\mathbf{v_1}$, $\mathbf{v}_\perp$ and then apply the trigonometric identity $\mathbf{v} = \mathbf{v_1}\cos\theta' + \mathbf{v}_\perp \sin\theta'$.

$$slerp(\mathbf{q}_1, \mathbf{q}_2, t) = \frac{\mathbf{q}_1 \sin((1-t)\theta) + \mathbf{q}_2 \sin(t\theta)}{\sin\theta}.$$

An intuitive way to understand `Slerp()` was presented by Blow (2004). As context, given the quaternions to interpolate between, $\mathbf{q}_1$ and $\mathbf{q}_2$, denote by θ the angle between them. Then, given a parameter value $t \in [0, 1]$, we'd like to find the intermediate quaternion $\mathbf{q}'$ that makes angle $\theta' = \theta t$ between it and $\mathbf{q}_1$, along the path from $\mathbf{q}_1$ to $\mathbf{q}_2$.

An easy way to compute $\mathbf{q}'$ is to first compute an orthogonal coordinate system in the space of quaternions where one axis is $\mathbf{q}_1$ and the other is a quaternion orthogonal to $\mathbf{q}_1$ such that the two axes form a basis that spans $\mathbf{q}_1$ and $\mathbf{q}_2$. Given such a coordinate system, we can compute rotations with respect to $\mathbf{q}_1$. (See Figure 2.16, which illustrates the concept in the 2D setting.) An orthogonal vector $\mathbf{q}_\perp$ can be found by projecting $\mathbf{q}_1$ onto $\mathbf{q}_2$ and then subtracting the orthogonal projection from $\mathbf{q}_2$; the remainder is guaranteed to be orthogonal to $\mathbf{q}_1$:

$$\mathbf{q}_\perp = \mathbf{q}_2 - (\mathbf{q}_1 \cdot \mathbf{q}_2)\mathbf{q}_1. \tag{2.7}$$

Given the coordinate system, quaternions along the animation path are given by

$$\mathbf{q}' = \mathbf{q}_1 \cos(\theta t) + \mathbf{q}_\perp \sin(\theta t). \tag{2.8}$$

The implementation of the `Slerp()` function checks to see if the two quaternions are nearly parallel, in which case it uses regular linear interpolation of quaternion components in order to avoid numerical instability. Otherwise, it computes an orthogonal quaternion `qperp` using Equation (2.7) and then computes the interpolated quaternion with Equation (2.8).

`Slerp()` 96

⟨*Quaternion Method Definitions*⟩ ≡
```
Quaternion Slerp(float t, const Quaternion &q1,
                 const Quaternion &q2) {
    float cosTheta = Dot(q1, q2);
    if (cosTheta > .9995f)
        return Normalize((1.f - t) * q1 + t * q2);
    else {
        float theta = acosf(Clamp(cosTheta, -1.f, 1.f));
        float thetap = theta * t;
        Quaternion qperp = Normalize(q2 - q1 * cosTheta);
        return q1 * cosf(thetap) + qperp * sinf(thetap);
    }
}
```

2.9.3 AnimatedTransform IMPLEMENTATION

Given the foundations of the quaternion infrastructure, we can now implement the AnimatedTransform class, which implements keyframe transformation interpolation in pbrt. Its constructor takes two transformations and the time values they are associated with.

Following an approach introduced by Shoemake and Duff (1992), AnimatedTransform decomposes the given composite transformation matrices into scaling, rotation, and translation components. The decomposition is performed by the AnimatedTransform::Decompose() method.

⟨*AnimatedTransform Public Methods*⟩ ≡
```
AnimatedTransform(const Transform *transform1, float time1,
                  const Transform *transform2, float time2)
    : startTime(time1), endTime(time2),
      startTransform(transform1), endTransform(transform2),
      actuallyAnimated(*startTransform != *endTransform) {
    Decompose(startTransform->m, &T[0], &R[0], &S[0]);
    Decompose(endTransform->m, &T[1], &R[1], &S[1]);
}
```

⟨*AnimatedTransform Private Data*⟩ ≡
```
const float startTime, endTime;
const Transform *startTransform, *endTransform;
const bool actuallyAnimated;
Vector T[2];
Quaternion R[2];
Matrix4x4 S[2];
```

AnimatedTransform 96
AnimatedTransform::Decompose() 97
Clamp() 1000
Matrix4x4 1021
Quaternion 92
Quaternion::Dot() 93
Quaternion::Normalize() 93
Transform 76
Vector 57

Given the composite matrix for a transformation, information has been lost about any individual transformations that were composed to compute it. For example, given the matrix for the product of a translation and then a scale, an equal matrix could also be computed by first scaling and then translating (by different amounts). Thus, we need to choose a canonical sequence of transformations for the decomposition. For our needs

here, this decision isn't significant. (It would be more important in an animation system that was decomposing composite transformations in order to make them editable by changing individual components, for example.)

Further, we will handle only affine transformations here, which is what is needed for animating cameras and geometric primitives in a rendering system; perspective transformations aren't generally relevant to animation of objects like these.

The transformation decomposition we will use is the following:

$$\mathbf{M} = \mathbf{TRS},$$

where $\mathbf{M}$ is the given transformation, $\mathbf{T}$ is a translation, $\mathbf{R}$ is a rotation, and $\mathbf{S}$ is a scale. $\mathbf{S}$ is actually a generalized scale (Shoemake and Duff call it *stretch*) that represents a scale in *some* coordinate system, just not necessarily the current one. In any case, it can still be properly interpolated with linear interpolation of its components. The `Decompose()` method computes the decomposition given a `Matrix4x4`.

⟨*AnimatedTransform Method Definitions*⟩ ≡
```
void AnimatedTransform::Decompose(const Matrix4x4 &m, Vector *T,
                                  Quaternion *Rquat, Matrix4x4 *S) {
    ⟨Extract translation T from transformation matrix 97⟩
    ⟨Compute new transformation matrix M without translation 97⟩
    ⟨Extract rotation R from transformation matrix 98⟩
    ⟨Compute scale S using rotation and original matrix 98⟩
}
```

Extracting the translation $\mathbf{T}$ is easy; it can be found directly from the appropriate elements of the 4×4 transformation matrix.

⟨*Extract translation* `T` *from transformation matrix*⟩ ≡ 97
```
T->x = m.m[0][3];
T->y = m.m[1][3];
T->z = m.m[2][3];
```

Since we are assuming an affine transformation (no projective components), after we remove the translation, what is left is the upper 3×3 matrix that represents scaling and rotation together. This matrix is copied into a new matrix `M` for further processing.

⟨*Compute new transformation matrix* `M` *without translation*⟩ ≡ 97
```
Matrix4x4 M = m;
for (int i = 0; i < 3; ++i)
    M.m[i][3] = M.m[3][i] = 0.f;
M.m[3][3] = 1.f;
```

Next we'd like to extract the pure rotation component of `M`. We'll use a technique called *polar decomposition* to do this. It can be shown that the polar decomposition of a matrix $\mathbf{M}$ into rotation $\mathbf{R}$ and scale $\mathbf{S}$ can be computed by successively averaging $\mathbf{M}$ to its inverse transpose

$$\mathbf{M}_{i+1} = \frac{1}{2}\left(\mathbf{M}_i + (\mathbf{M}_i^T)^{-1}\right) \quad [2.9]$$

Matrix4x4 1021
Matrix4x4::m 1021
Quaternion 92
Vector 57

until convergence. (It's easy to see that if **M** is a pure rotation, then averaging it with its inverse transpose will leave it unchanged, since its inverse is equal to its transpose. The "Further Reading" section has more references that discuss why this series converges to the rotation component of the original transformation.) Shoemake and Duff (1992) proved that the resulting matrix is the closest orthogonal matrix to **M**—a desirable property.

To compute this series, we iteratively apply Equation (2.9) until either the difference between successive terms is small or a fixed number of iterations have been performed. In practice, this series generally converges quickly.

⟨*Extract rotation* R *from transformation matrix*⟩ ≡ 97
```
float norm;
int count = 0;
Matrix4x4 R = M;
do {
    ⟨Compute next matrix Rnext in series 98⟩
    ⟨Compute norm of difference between R and Rnext 98⟩
    R = Rnext;
} while (++count < 100 && norm > .0001f);
*Rquat = Quaternion(R);
```

⟨*Compute next matrix* Rnext *in series*⟩ ≡ 98
```
Matrix4x4 Rnext;
Matrix4x4 Rit = Inverse(Transpose(R));
for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 4; ++j)
        Rnext.m[i][j] = 0.5f * (R.m[i][j] + Rit.m[i][j]);
```

⟨*Compute norm of difference between* R *and* Rnext⟩ ≡ 98
```
norm = 0.f;
for (int i = 0; i < 3; ++i) {
    float n = fabsf(R.m[i][0] - Rnext.m[i][0]) +
              fabsf(R.m[i][1] - Rnext.m[i][1]) +
              fabsf(R.m[i][2] - Rnext.m[i][2]);
    norm = max(norm, n);
}
```

Once we've extracted the rotation from **M**, the scale is all that's left. We would like to find the matrix **S** that satisfies $\mathbf{M} = \mathbf{RS}$. Now that we know both **R** and **M**, we just solve for $\mathbf{S} = \mathbf{R}^{-1}\mathbf{M}$.

⟨*Compute scale* S *using rotation and original matrix*⟩ ≡ 97
```
*S = Matrix4x4::Mul(Inverse(R), M);
```

The `Interpolate()` method computes the interpolated transformation matrix at a given time. The matrix is found by interpolating the previously extracted translation, rotation, and scale and then multiplying them together to get a composite matrix that represents the effect of the three transformations together.

Inverse() 1021
Matrix4x4 1021
Matrix4x4::m 1021
Matrix4x4::Mul() 1021
Quaternion 92
Transpose() 1021

⟨*AnimatedTransform Method Definitions*⟩ +≡
```
void AnimatedTransform::Interpolate(float time, Transform *t) const {
    ⟨Handle boundary conditions for matrix interpolation 99⟩
    float dt = (time - startTime) / (endTime - startTime);
    ⟨Interpolate translation at dt 99⟩
    ⟨Interpolate rotation at dt 99⟩
    ⟨Interpolate scale at dt 99⟩
    ⟨Compute interpolated matrix as product of interpolated components 100⟩
}
```

If the given time value is outside the time range of the two transformations stored in the `AnimatedTransform`, then the transformation at the start time or end time is returned, as appropriate. The `AnimatedTransform` constructor also checks whether the two `Transforms` stored are the same; if so, then no interpolation is necessary either. Most of the classes in `pbrt` that support animation always store an `AnimatedTransform` for their transformation, rather than storing either a `Transform` or `AnimatedTransform` as appropriate. This simplifies their implementations, though it does make it worthwhile to check for this case here and not unnecessarily do the work to interpolate between two equal transformations.

⟨*Handle boundary conditions for matrix interpolation*⟩ ≡ 99
```
if (!actuallyAnimated || time <= startTime) {
    *t = *startTransform;
    return;
}
if (time >= endTime) {
    *t = *endTransform;
    return;
}
```

The `dt` variable stores the offset in the range from `startTime` to `endTime`; it is zero at `startTime` and one at `endTime`. Given `dt`, interpolation of the translation is trivial.

AnimatedTransform 96
AnimatedTransform::actuallyAnimated 96
AnimatedTransform::endTime 96
AnimatedTransform::endTransform 96
AnimatedTransform::R 96
AnimatedTransform::S 96
AnimatedTransform::startTime 96
AnimatedTransform::startTransform 96
AnimatedTransform::T 96
Matrix4x4 1021
Quaternion 92
Slerp() 96
Transform 76
Vector 57

⟨*Interpolate translation at* `dt`⟩ ≡ 99
```
Vector trans = (1.f - dt) * T[0] + dt * T[1];
```

The rotation is interpolated between the start and end rotations using the `Slerp()` routine (Section 2.9.2).

⟨*Interpolate rotation at* `dt`⟩ ≡ 99
```
Quaternion rotate = Slerp(dt, R[0], R[1]);
```

Finally, the interpolated scale matrix is computed by just interpolating the individual elements of the start and end scale matrices. Because the `Matrix4x4` constructor sets the matrix to the identity matrix, we don't need to initialize any of the other elements.

⟨*Interpolate scale at* `dt`⟩ ≡ 99
```
Matrix4x4 scale;
for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 3; ++j)
        scale.m[i][j] = Lerp(dt, S[0].m[i][j], S[1].m[i][j]);
```

Given the three interpolated parts, the product of their three transformation matrices gives us the final result.

⟨*Compute interpolated matrix as product of interpolated components*⟩ ≡ 99
```
*t = Translate(trans) * rotate.ToTransform() * Transform(scale);
```

AnimatedTransform also provides a number of methods that apply interpolated transformations directly, using the provided time for Points and Vectors and Ray::time for Rays. These methods are more efficient than calling AnimatedTransform::Interpolate() and then using the returned matrix when there is no actual animation, since a copy of the transformation matrix doesn't need to be made.

⟨*AnimatedTransform Public Methods*⟩ +≡
```
void operator()(const Ray &r, Ray *tr) const;
void operator()(const RayDifferential &r, RayDifferential *tr) const;
Point operator()(float time, const Point &p) const;
Vector operator()(float time, const Vector &v) const;
Ray operator()(const Ray &r) const;
```

Given a geometric primitive with an animated transformation, it's necessary to be able to compute a bounding box that encompasses the primitive's motion over the animation time period. The AnimatedTransform::MotionBounds() method takes a bounding box and computes a bounding box of its animated motion over the AnimatedTransform's time range. Its useInverse parameter indicates whether the forward or inverse transform should be used to transform the bounding box.

This method, which is based on computing the animated transformation at a large number of time samples, may compute an insufficiently conservative bounding box as implemented here (though in practice it works well). An exercise at the end of the chapter discusses its limitations in more detail and suggests a direction for a more robust approach.

⟨*AnimatedTransform Method Definitions*⟩ +≡
```
BBox AnimatedTransform::MotionBounds(const BBox &b,
                                     bool useInverse) const {
    if (!actuallyAnimated) return Inverse(*startTransform)(b);
    BBox ret;
    const int nSteps = 128;
    for (int i = 0; i < nSteps; ++i) {
        Transform t;
        float time = Lerp(float(i)/float(nSteps-1), startTime, endTime);
        Interpolate(time, &t);
        if (useInverse) t = Inverse(t);
        ret = Union(ret, t(b));
    }
    return ret;
}
```

AnimatedTransform 96
AnimatedTransform::actuallyAnimated 96
AnimatedTransform::endTime 96
AnimatedTransform::Interpolate() 99
AnimatedTransform::MotionBounds() 100
AnimatedTransform::startTime 96
AnimatedTransform::startTransform 96
BBox 70
Point 63
Quaternion::ToTransform() 94
Ray 66
Ray::time 67
RayDifferential 69
Transform 76
Transform::Inverse() 77
Transform::operator() 86
Translate() 79
Union() 72
Vector 57

2.10 DIFFERENTIAL GEOMETRY

We will wrap up this chapter by developing a self-contained representation for the geometry of a particular point on a surface (typically the point of a ray intersection). This abstraction needs to hide the particular type of geometric shape the point lies on, supplying enough information about the surface point to allow the shading and geometric operations in the rest of `pbrt` to be implemented generically, without the need to distinguish between different shape types such as spheres and triangles. The `DifferentialGeometry` class implements just such an abstraction; its implementation is in the files `core/diffgeom.h` and `core/diffgeom.cpp`.

The information needed by the rest of the system includes:

- The position of the 3D point p
- The surface normal $\mathbf{n}$ at the point
- (u, v) coordinates from the parameterization of the surface
- The parametric partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$
- The partial derivatives of the change in surface normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$
- A pointer to the `Shape` that the differential geometry lies on (the `Shape` class will be introduced in the next chapter)

See Figure 2.17 for a depiction of these values.

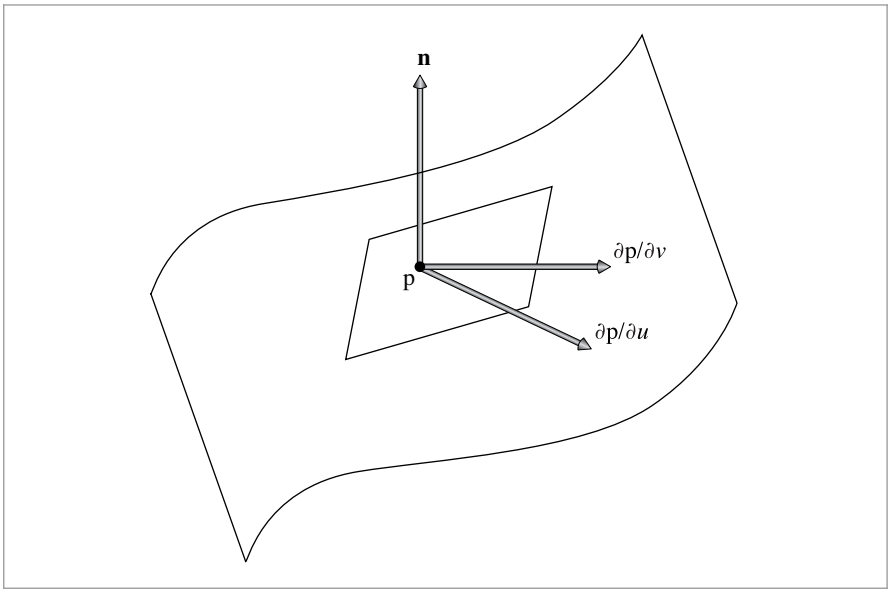


Figure 2.17: The Local Differential Geometry around a Point p. The parametric partial derivatives of the surface, $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$, lie in the tangent plane but are not necessarily orthogonal. The surface normal $\mathbf{n}$ is given by the cross product of $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$. The vectors $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ (not shown here) record the differential change in surface normal as we move u and v along the surface.

`DifferentialGeometry` 102

`Shape` 108

This representation implicitly assumes that shapes have a parametric description—that for some range of (u, v) values, points on the surface are given by some function f such that $p = f(u, v)$. Although this isn't true for all shapes, all of the shapes that pbrt supports do have at least a local parametric description, so we will stick with the parametric representation since this assumption is helpful elsewhere (e.g., for antialiasing of textures in Chapter 10).

⟨*DifferentialGeometry Declarations*⟩ ≡
```
struct DifferentialGeometry {
    DifferentialGeometry() { u = v = 0.; shape = NULL; }
    ⟨DifferentialGeometry Public Methods⟩
    ⟨DifferentialGeometry Public Data 102⟩
};
```

The names of the member variables directly correspond to the quantities they represent. Here we have named the `Normal` member variable `nn` with a second "n" to reflect the fact that it is a normalized version of the surface normal.

⟨*DifferentialGeometry Public Data*⟩ ≡ 102
```
Point p;
Normal nn;
float u, v;
const Shape *shape;
```

We also need to store the partial derivatives of the surface position and the surface normal:

⟨*DifferentialGeometry Public Data*⟩ +≡ 102
```
Vector dpdu, dpdv;
Normal dndu, dndv;
```

The `DifferentialGeometry` constructor only needs a few parameters—the point of interest, the partial derivatives of position and normal, and the (u, v) coordinates. It computes the normal as the cross product of the partial derivatives.

⟨*DifferentialGeometry Method Definitions*⟩ ≡
```
DifferentialGeometry::DifferentialGeometry(const Point &P,
        const Vector &DPDU, const Vector &DPDV,
        const Normal &DNDU, const Normal &DNDV,
        float uu, float vv, const Shape *sh)
    : p(P), dpdu(DPDU), dpdv(DPDV), dndu(DNDU), dndv(DNDV) {
    ⟨Initialize DifferentialGeometry from parameters 102⟩
    ⟨Adjust normal based on orientation and handedness 103⟩
}
```

⟨*Initialize* `DifferentialGeometry` *from parameters*⟩ ≡ 102
```
nn = Normal(Normalize(Cross(dpdu, dpdv)));
u = uu;
v = vv;
shape = sh;
```

Cross() 62
DifferentialGeometry 102
DifferentialGeometry::nn 102
DifferentialGeometry::shape 102
DifferentialGeometry::u 102
DifferentialGeometry::v 102
Normal 65
Point 63
Shape 108
Vector 57
Vector::Normalize() 63

The surface normal has special meaning to pbrt, which assumes that, for closed shapes, the normal is oriented such that it points to the "outside" of the shape. For example, this assumption will be used later when we need to decide if a ray is entering or leaving the volume enclosed by a shape. Furthermore, for geometry used as an area light source, light is emitted from only the side of the surface that the normal points toward; the other side is black.

Because normals have this special meaning, pbrt provides a mechansim for the user to reverse the orientation of the normal, flipping it to point in the opposite direction. The ReverseOrientation directive in pbrt's input file flips the normal to point in the opposite, nondefault direction. Therefore, it will be necessary to check if the given Shape has this flag set and, if so, switch the normal's direction.

However, one other factor plays into the orientation of the normal and must be accounted for here as well. If the Shape's transformation matrix has switched the handedness of the object coordinate system from pbrt's default left-handed coordinate system to a right-handed one, we need to switch the orientation of the normal as well. To see why this is so, consider a scale matrix $\mathbf{S}(1, 1, -1)$. We would naturally expect this scale to switch the direction of the normal, although because we compute the normal by $\mathbf{n} = \partial \mathrm{p}/\partial u \times \partial \mathrm{p}/\partial v$ it can be shown that:

$$\begin{aligned}\mathbf{S}(1, 1, -1)\frac{\partial \mathrm{p}}{\partial u} \times \mathbf{S}(1, 1, -1)\frac{\partial \mathrm{p}}{\partial v} &= \mathbf{S}(-1, -1, 1)\frac{\partial \mathrm{p}}{\partial u} \times \frac{\partial \mathrm{p}}{\partial v} \\ &= \mathbf{S}(-1, -1, 1)\mathbf{n} \neq \mathbf{S}(1, 1, -1)\mathbf{n}.\end{aligned}$$

Therefore, it is also necessary to manually flip the normal's direction if the transformation switches the handedness of the coordinate system, since the flip won't be accounted for by the computation of the normal's direction using the cross product.

The normal's direction is swapped if one but not both of these two conditions is met; if both were met, their effect would cancel out. The exclusive-OR operation tests this condition.

⟨*Adjust normal based on orientation and handedness*⟩ ≡ 102

```
if (shape && (shape->ReverseOrientation ^ shape->TransformSwapsHandedness))
    nn *= -1.f;
```

FURTHER READING

DeRose, Goldman, and their collaborators have argued for an elegant "coordinate-free" approach to describing vector geometry for graphics, where the fact that positions and directions happen to be represented by (x, y, z) coordinates with respect to a particular coordinate system is deemphasized and where points and vectors themselves record which coordinate system they are expressed in terms of (Goldman 1985; DeRose 1989; Mann, Litke, and DeRose 1997). This makes it possible for a software layer to ensure that common errors like adding a vector in one coordinate system to a point in another coordinate system are transparently handled by transforming them to a common coordinate system first. We have not followed this approach in pbrt, although the principles behind

DifferentialGeometry::nn 102
DifferentialGeometry::shape 102
Shape::ReverseOrientation 108
Shape::TransformSwapsHandedness 108

this approach are well worth understanding and keeping in mind when working with coordinate systems in computer graphics.

Schneider and Eberly's *Geometric Tools for Computer Graphics* is influenced by the coordinate-free approach and covers the topics of this chapter in much greater depth (Schneider and Eberly 2003). It is also full of useful geometric algorithms for graphics. A classic and more traditional introduction to the topics of this chapter is *Mathematical Elements for Computer Graphics* by Rogers and Adams (1990). Note that their book uses a row-vector representation of points and vectors, however, which means that our matrices would be transposed when expressed in their framework, and that they multiply points and vectors by matrices to transform them ($\mathbf{pM}$), rather than multiplying matrices by points as we do ($\mathbf{Mp}$). Homogeneous coordinates were only briefly mentioned in this chapter, although they are the basis of projective geometry, where they are the foundation of many elegant algorithms. Stolfi's book is an excellent introduction to this topic (Stolfi 1991).

There are many good books on linear algebra and vector geometry. We have found Lang (1986) and Buck (1978) to be good references on these respective topics. See also Akenine-Möller et al.'s *Real-Time Rendering* book (2008) for a solid graphics-based introduction to linear algebra.

The subtleties of how normal vectors are transformed were first widely understood in the graphics community after articles by Wallis (1990) and Turkowski (1990c).

Shoemake (1985) introduced quaternions to graphics and showed their utility for animating rotations. Using polar matrix decomposition for animating transformations was described by Shoemake and Duff (1992); Higham (1986) developed the algorithm for extracting the rotation from a composite rotation and scale matrix by successively adding the matrix to its inverse transpose. Shoemake's chapters in *Graphics Gems* (1991, 1994a) respectively give more details on the derivation of the conversion from matrices to quaternions and the implementation of polar matrix decomposition.

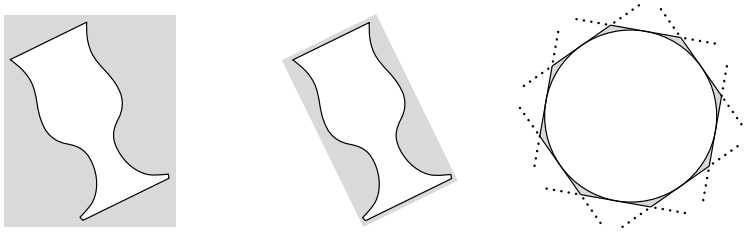
We followed Blow's derivation of spherical linear interpolation (2004) in our exposition in this chapter. Bloom et al. (2004) discuss desirable properties of interpolation of rotations for animation in computer graphics and which approaches deliver which of these properties. For more sophisticated approaches to rotation interpolation, see Ramamoorthi and Barr (1997) and Buss and Fillmore (2001).

EXERCISES

❶ 2.1 Find a more efficient way to transform axis-aligned bounding boxes by taking advantage of the symmetries of the problem: because the eight corner points are linear combinations of three axis-aligned basis vectors and a single corner point, their transformed bounding box can be found much more efficiently than by the method we presented (Arvo 1990).

❷ 2.2 Reimplement the core geometry classes using a vector instruction set such as Intel's SSE. Evaluate the performance implications on scenes with a variety of characteristics. What can you conclude about the effectiveness of this approach?

2 2.3 Instead of boxes, tighter bounds around objects could be computed by using the intersections of many nonorthogonal slabs. Extend the bounding box class in `pbrt` to allow the user to specify a bound comprised of arbitrary slabs.



Axis-aligned bounding box | Non-axis-aligned bounding box | Arbitrary bounding slabs

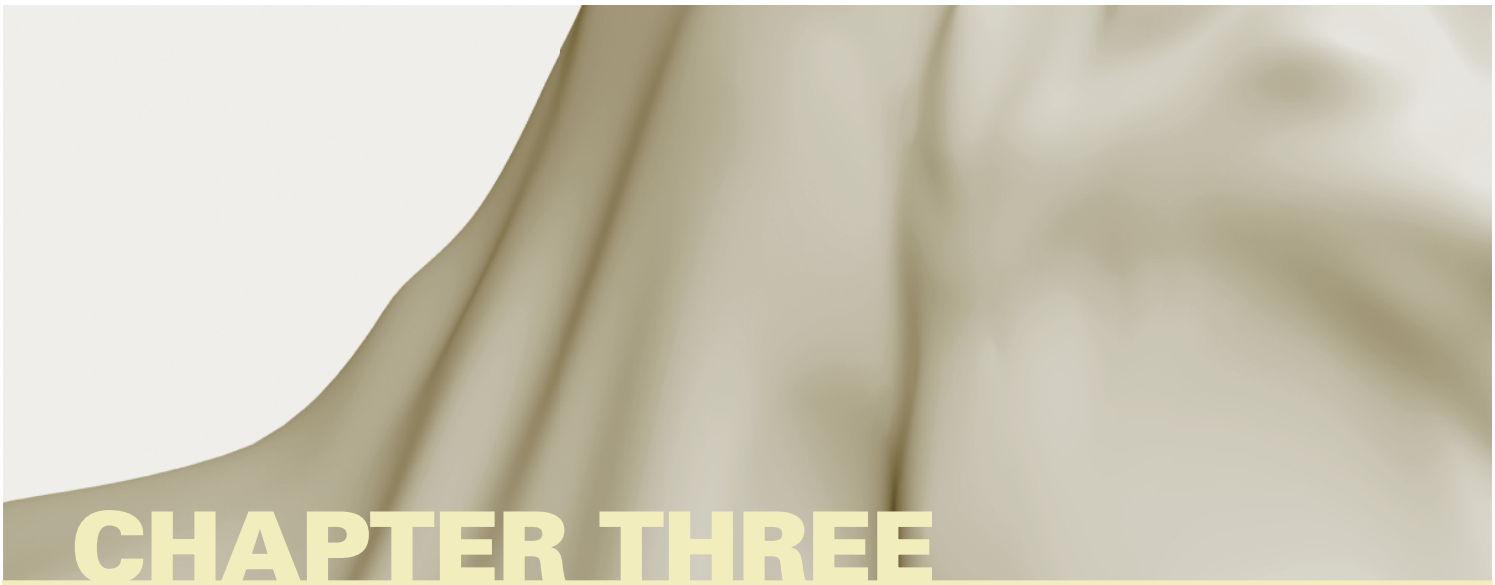
1 2.4 Change `pbrt` so that it transforms `Normals` just like `Vectors` and create a scene that gives a clearly incorrect image due to this bug. (Don't forget to eliminate this change from your copy of the source code when you're done!)

2 2.5 If only the translation components of a transformation are time varying, for example, then the `AnimatedTransform` implementation does unnecessary computation in interpolating between two rotations that are the same. Modify the `AnimatededTransform` impementation so that it avoids this work in cases where the full generality of its current implementation isn't necessary. How much of a performance difference do you observe for scenes where your optimizations are applicable?

3 2.6 The `AnimatedTransform::MotionBounds()` method is not guaranteed to be robust: because it samples the animated transform at a number of discrete times, it's not guaranteed that it will locate the extrema of motion of the bounding box and thus the computed bound may not actually bound the full motion of the object. Can you derive a more robust method that gives *guaranteed* conservative bounds?

A reasonable approach might be to write out formulas that give the (x, y, z) components of points as they are animated as a function of time, in terms of the values of decomposed matrix components. The extrema of each of these will come when their derivative with respect to time is zero; given the coordinates of the corners of the bounding box, is it possible to solve for the time values where these positions reach extrema? If it's not possible to solve for them in closed form, can you guide sampling of the parameter space based on a search for zero derivatives?

AnimatedTransform 96
AnimatedTransform::
MotionBounds() 100
Normal 65
Vector 57



CHAPTER THREE

03 SHAPES

In this chapter, we will present pbrt's abstraction for geometric primitives such as spheres and triangles. Careful abstraction of geometric shapes in a ray tracer is a key component of a clean system design, and shapes are the ideal candidate for an object-oriented approach. All geometric primitives implement a common interface, and the rest of the renderer can use this interface without needing any details about the underlying shape. This makes it possible to isolate the geometric and shading subsystems of pbrt. Without this isolation, adding new shapes to the system would be unnecessarily difficult and error prone.

pbrt hides details about its primitives behind a two-level abstraction. The Shape class provides access to the raw geometric properties of the primitive, such as its surface area and bounding box, and provides a ray intersection routine. The Primitive class provides additional nongeometric information about the primitive, such as its material properties. The rest of the renderer then deals only with the abstract Primitive interface. This chapter will focus on the geometry-only Shape class; the Primitive interface is the topic of Chapter 4.

The interface for Shapes is in the source file core/shape.h, and definitions of common Shape methods can be found in core/shape.cpp.

3.1 BASIC SHAPE INTERFACE

The Shape class in pbrt is *reference counted*—pbrt keeps track of the number of outstanding pointers to a particular shape and automatically deletes the shape when that reference count goes to zero. Although not completely foolproof, this is a form of *garbage collection* that eliminates the need to have to worry about freeing shape memory at the wrong

time. The ReferenceCounted class handles all of the underlying mechanisms; its implementation is in Section A.5.2 in Appendix A. Reference counted objects, like Shape, must inherit from the ReferenceCounted base class.

⟨*Shape Declarations*⟩ ≡
```
class Shape : public ReferenceCounted {
public:
    ⟨Shape Interface 109⟩
    ⟨Shape Public Data 108⟩
};
```

All shapes are defined in object coordinate space; for example, all spheres are defined in a coordinate system where the center of the sphere is at the origin. In order to place a sphere at another position in the scene, a transformation that describes the mapping from object space to world space must be provided. The Shape class stores both this transformation and its inverse. Shapes also take a Boolean parameter, ReverseOrientation, that indicates whether their surface normal directions should be reversed from the default. This capability is useful because the orientation of the surface normal is used to determine which side of a shape is "outside." For example, shapes that emit illumination are emissive only on the side the surface normal lies on. The value of this parameter is set via the ReverseOrientation statement in pbrt input files.

Shapes also store the result of the Transform::SwapsHandedness() call for their object-to-world transformation. This value is needed by the DifferentialGeometry constructor that will be called each time a ray intersection is found, so the Shape constructor computes it once and stores it.

⟨*Shape Method Definitions*⟩ ≡
```
Shape::Shape(const Transform *o2w, const Transform *w2o, bool ro)
    : ObjectToWorld(o2w), WorldToObject(w2o), ReverseOrientation(ro),
      TransformSwapsHandedness(o2w->SwapsHandedness()),
      shapeId(nextshapeId++) {
}
```

An important detail is that shapes store pointers to their transformations rather than Transform objects directly. Recall from Section 2.7 that Transform objects require 32 floats, requiring 128 bytes of memory; because multiple shapes in the scene will frequently have the same transformation applied to them, pbrt keeps a pool of Transforms so that they can be re-used and passes pointers to the shared Transforms to the shapes. As such, the Shape destructor does not delete its Transform pointers, leaving the Transform management code to manage that memory instead.

⟨*Shape Public Data*⟩ ≡ 108
```
const Transform *ObjectToWorld, *WorldToObject;
const bool ReverseOrientation, TransformSwapsHandedness;
```

All Shapes in the system are given a unique 32-bit numeric id, stored here in the shapeId member variable. This identifier has a variety of uses, among them the adaptive image sampling routines that take additional samples in the areas of pixels that have multiple shapes overlapping them—pixels with more geometric complexity are likely to require

ReferenceCounted 1010
Shape 108
Transform 76
Transform::
SwapsHandedness() 89

higher sampling rates than those with less for a high-quality final image. Having this identifier available makes it easy for those routines to detect this situation.

⟨*Shape Public Data*⟩ +≡ 108
```
const uint32_t shapeId;
static uint32_t nextshapeId;
```

The first value assigned for a shape id is one, so that the value of zero for a shape id can be used to indicate "no shape" (for example, as the default value in the `Intersection` structure, which stores the id of the shape that a ray hits).

⟨*Shape Method Definitions*⟩ +≡
```
uint32_t Shape::nextshapeId = 1;
```

3.1.1 BOUNDING

Each `Shape` subclass must be capable of bounding itself with a `BBox`. There are two different bounding methods. The first, `ObjectBound()`, returns a bounding box in the shape's object space, and the second, `WorldBound()`, returns a bounding box in world space. The implementation of the first method is left up to each individual shape, but there is a default implementation of the second method that transforms the object bound to world space. Shapes that can easily compute a tighter world space bound should override this method, however. An example of such a shape is a triangle (Figure 3.1).

⟨*Shape Interface*⟩ ≡ 108
```
virtual BBox ObjectBound() const = 0;
```

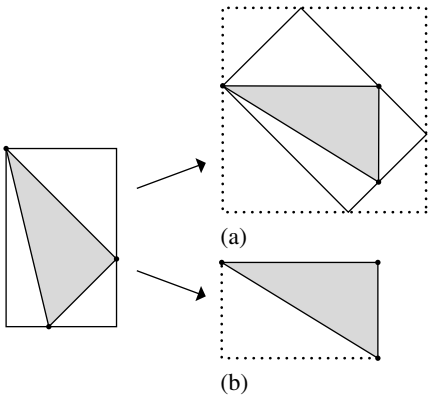


Figure 3.1: A world space bounding box of a triangle is computed by transforming its object space bounding box to world space and then finding the bounding box that encloses the resulting bounding box; a sloppy bound may result. (a) However, if the triangle's vertices are first transformed from object space to world space and then bounded (b), the fit of the bounding box can be much better.

`BBox` 70
`Intersection` 186
`Shape` 108

⟨*Shape Method Definitions*⟩ +≡
```
BBox Shape::WorldBound() const {
    return (*ObjectToWorld)(ObjectBound());
}
```

3.1.2 REFINEMENT

Not every shape needs to be capable of determining whether a ray intersects it. For example, a complex surface might first be tessellated into triangles, which can then be intersected directly. Another possibility is a shape that is a placeholder for a large amount of geometry stored on disk. We could store just the filename of the geometry file and the bounding box of the geometry in memory, and read the geometry in from disk only if a ray pierces the bounding box.

The default implementation of the `Shape::CanIntersect()` function indicates that a shape *can* compute ray intersections, so only shapes that are nonintersectable need to override this method.

⟨*Shape Method Definitions*⟩ +≡
```
bool Shape::CanIntersect() const {
    return true;
}
```

If the shape cannot be intersected directly, it must provide a `Shape::Refine()` method that splits the shape into a group of new shapes, some of which may be intersectable and some of which may need further refinement. The default implementation of the `Shape::Refine()` method issues an error message; thus, shapes that are intersectable (which is the common case) do not have to provide an empty instance of this method. `pbrt` will never call `Shape::Refine()` if `Shape::CanIntersect()` returns `true`.

⟨*Shape Method Definitions*⟩ +≡
```
void Shape::Refine(vector<Reference<Shape> > &refined) const {
    Severe("Unimplemented Shape::Refine() method called");
}
```

3.1.3 INTERSECTION

The `Shape` class provides two intersection routines. The first, `Shape::Intersect()`, returns geometric information about a single ray-shape intersection corresponding to the first intersection, if any, in the `[mint, maxt]` parametric range along the ray. The other, `Shape::IntersectP()`, is a predicate function that determines whether or not an intersection occurs, without returning any details about the intersection itself. Most shape implementations can provide a more efficient implementation for `IntersectP()` that can determine whether an intersection exists without computing all of its details.

There are a few important things to keep in mind when reading (and writing) intersection routines:

- The `Ray` structure contains `Ray::mint` and `Ray::maxt` variables that define a ray *segment*. Intersection routines must ignore any intersections that do not occur along this segment.

BBox 70
Ray 66
Ray::maxt 67
Ray::mint 67
Reference 1011
Severe() 1005
Shape 108
Shape::CanIntersect() 110
Shape::Intersect() 111
Shape::IntersectP() 111
Shape::ObjectToWorld 108
Shape::Refine() 110

- If an intersection is found, its parametric distance along the ray should be stored in the pointer `tHit` that is passed into the intersection routine. If there are multiple intersections along the ray, the closest one should be reported.
- If an intersection is found, the value pointed to by the `rayEpsilon` parameter must be initialized with a value that describes the maximum numeric error in the intersection calculation. This parameter is discussed further in Section 3.1.4.
- Information about an intersection is stored in the `DifferentialGeometry` structure, which completely captures the local geometric properties of a surface. This class is used heavily throughout `pbrt`, and it serves to cleanly isolate the geometric portion of the ray tracer from the shading and illumination portions. The `DifferentialGeometry` class was defined in Section 2.10.[1]
- The rays passed into intersection routines are in world space, so shapes are responsible for transforming them to object space if needed for intersection tests. The differential geometry returned should be in world space.

Rather than making the intersection routines pure virtual functions, the `Shape` class provides default implementations of the intersect routines that print an error message if they are called. All `Shapes` that return `true` from `Shape::CanIntersect()` must provide implementations of these functions; those that return `false` can depend on `pbrt` not to call these routines on nonintersectable shapes. If these were pure virtual functions, then each nonintersectable shape would have to implement a similar default function.

⟨*Shape Method Definitions*⟩ +≡
```
bool Shape::Intersect(const Ray &ray, float *tHit, float *rayEpsilon,
                      DifferentialGeometry *dg) const {
    Severe("Unimplemented Shape::Intersect() method called");
    return false;
}
```

⟨*Shape Method Definitions*⟩ +≡
```
bool Shape::IntersectP(const Ray &ray) const {
    Severe("Unimplemented Shape::IntersectP() method called");
    return false;
}
```

3.1.4 AVOIDING INCORRECT SELF-INTERSECTIONS

A classic issue in ray tracing is the self-intersection problem, where floating-point errors can lead to incorrect self-intersections when rays are traced starting from a previously computed ray-surface intersection point (e.g., shadow rays or rays for specular reflection and refraction.). Figure 3.2 illustrates the problem: Represented in finite-precision

`DifferentialGeometry` 102
`Ray` 66
`Severe()` 1005
`Shape` 108
`Shape::CanIntersect()` 110

1 Almost all ray tracers use this general idiom for returning geometric information about intersections with shapes. As an optimization, many will only partially initialize the intersection information when an intersection is found, storing just enough information so that the rest of the values can be computed later if actually needed. This approach saves work in the case where a closer intersection is later found with another shape. In our experience, the extra work to compute all the information isn't substantial, and for renderers that have complex scene data management algorithms (e.g., discarding geometry from main memory when too much memory is being used and writing it to disk), the deferred approach may fail because the shape is no longer in memory.

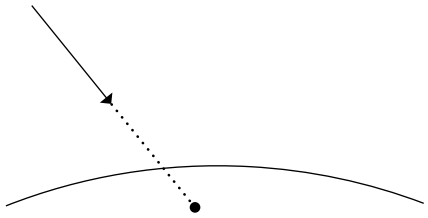


Figure 3.2: Due to finite floating-point precision and round-off error, when the intersection of a ray is found with a shape, the computed intersection point may lie slightly above or slightly below the true intersection point. This can lead to rendering errors when reflected and shadow rays are traced starting from the computed intersection point, as incorrect self-intersections with the surface may be detected.

floating-point numbers, the computed intersection point may lie slightly below the actual surface. If new rays are traced with this point as their origin, it's possible that they may intersect the surface even though a ray actually starting from the surface would not.

`pbrt` addresses this issue by requiring shape `Intersection()` method implementations to return a value through the `rayEpsilon` parameter that gives the maximum floating-point error in the parametric value computed for the intersection point. The integrators in Chapters 15 through 17 adjust the origins of rays spawned from intersection points based on this error term in order to ensure that they do not incorrectly reintersect the surface.

In general, computing this floating-point error bound requires careful numerical analysis of the specific computations performed in the particular ray–shape intersection algorithm used.

3.1.5 SHADING GEOMETRY

Some shapes (notably triangle meshes) support the idea of having two types of differential geometry at a point on the surface: the true geometry, which accurately reflects the local properties of the surface, and the *shading geometry*, which may have normals and tangents that are different than those in the true differential geometry. For triangle meshes, the user can provide normal vectors and primary tangents at the vertices of the mesh that are interpolated to give normals and tangents at points across the faces of triangles. Shading geometry with interpolated normals can make otherwise faceted triangle meshes appear to be more smooth than they geometrically are.

The `Shape::GetShadingGeometry()` method returns the shading geometry corresponding to the `DifferentialGeometry` returned by the `Shape::Intersect()` routine. By default, the shading geometry matches the true geometry, so the default implementation just copies the true geometry. One subtlety is that an object-to-world transformation is passed to this routine; if the routine needs to transform data from object space to world

DifferentialGeometry 102

space to compute the shading geometry, it must use this transformation rather than the Shape::ObjectToWorld transformation. This allows object instancing to be implemented in pbrt (see Section 4.1.2).

⟨*Shape Interface*⟩ +≡ 108
```
virtual void GetShadingGeometry(const Transform &obj2world,
        const DifferentialGeometry &dg,
        DifferentialGeometry *dgShading) const {
    *dgShading = dg;
}
```

3.1.6 SURFACE AREA

In order to properly use Shapes as area lights, it is necessary to be able to compute the surface area of a shape in object space. As with the intersection methods, this method will only be called for intersectable shapes.

⟨*Shape Method Definitions*⟩ +≡
```
float Shape::Area() const {
    Severe("Unimplemented Shape::Area() method called");
    return 0.;
}
```

3.1.7 SIDEDNESS

Many rendering systems, particularly those based on scan line or z-buffer algorithms, support the concept of shapes being "one-sided"—the shape is visible if seen from the front but disappears when viewed from behind. In particular, if a geometric object is closed and always viewed from the outside, then the back-facing parts of it can be discarded without changing the resulting image. This optimization can substantially improve the speed of these types of hidden surface removal algorithms. The potential for improved performance is reduced when using this technique with ray tracing, however, since it is often necessary to perform the ray-object intersection before determining the surface normal to do the back-facing test. Furthermore, this feature can lead to a physically inconsistent scene description if one-sided objects are not in fact closed. For example, a surface might block light when a shadow ray is traced from a light source to a point on another surface, but not if the shadow ray is traced in the other direction. For all of these reasons, pbrt doesn't support this feature.

3.2 SPHERES

Spheres are a special case of a general type of surface called *quadrics*—surfaces described by quadratic polynomials in x, y, and z. They are the simplest type of curved surface that is useful to a ray tracer and are a good starting point for general ray intersection routines. pbrt supports six types of quadrics: spheres, cones, disks (a special case of a cone), cylinders, hyperboloids, and paraboloids.

DifferentialGeometry 102
Severe() 1005
Shape::ObjectToWorld 108
Transform 76

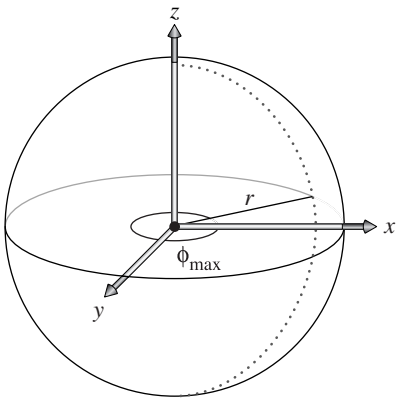


Figure 3.3: Basic Setting for the Sphere Shape. It has a radius of r and is centered at the object space origin. A partial sphere may be described by specifying a maximum ϕ value.

Many surfaces can be described in one of two main ways: in *implicit form* and in *parametric form*. An implicit function describes a 3D surface as

$$f(x, y, z) = 0.$$

The set of all points (x, y, z) that fulfill this condition defines the surface. For a unit sphere at the origin, the familiar implicit equation is $x^2 + y^2 + z^2 - 1 = 0$. Only the set of points one unit from the origin satisfies this constraint, giving the unit sphere's surface.

Many surfaces can also be described parametrically using a function to map 2D points to 3D points on the surface. For example, a sphere of radius r can be described as a function of 2D spherical coordinates (θ, ϕ), where θ ranges from 0 to π and ϕ ranges from 0 to 2π (Figure 3.3):

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta. \end{aligned}$$

We can transform this function $f(\theta, \phi)$ into a function $f(u, v)$ over $[0, 1]^2$ and also generalize it slightly to allow partial spheres that only sweep out $\theta \in [\theta_{min}, \theta_{max}]$ and $\phi \in [0, \phi_{max}]$ with the substitution

$$\begin{aligned} \phi &= u\,\phi_{max} \\ \theta &= \theta_{min} + v(\theta_{max} - \theta_{min}). \end{aligned}$$

This form is particularly useful for texture mapping, where it can be directly used to map a texture defined over $[0, 1]^2$ to the sphere. Figure 3.4 shows an image of two spheres; a grid image map has been used to show the (u, v) parameterization.

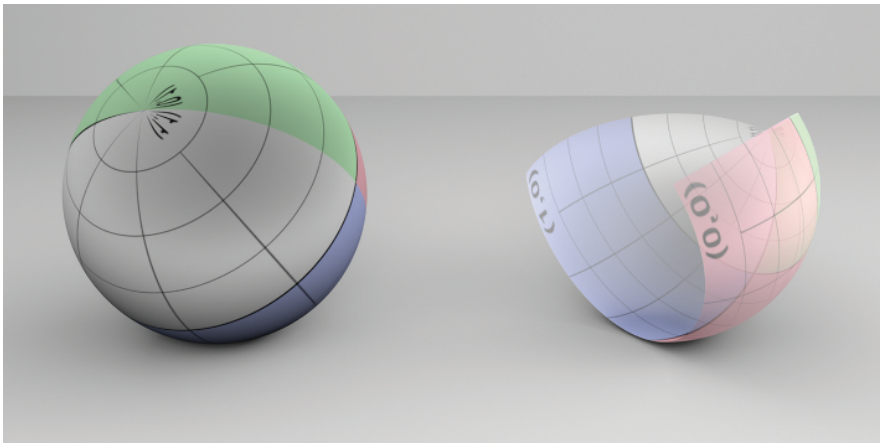


Figure 3.4: Two Spheres. On the left is a complete sphere, and on the right is a partial sphere (with $z_{\max} < r$ and $\phi_{\max} < 2\pi$). Note that the texture map used shows the (u, v) parameterization of the shape; the singularity at one of the poles is visible in the complete sphere.

As we describe the implementation of the sphere shape, we will make use of both the implicit and parametric descriptions of the shape, depending on which is a more natural way to approach the particular problem we're facing.

3.2.1 CONSTRUCTION

⟨*Sphere Declarations*⟩ ≡
```
class Sphere : public Shape {
public:
    ⟨Sphere Public Methods⟩
private:
    ⟨Sphere Private Data 116⟩
};
```

The `Sphere` class represents a sphere that is centered at the origin in object space. To place it elsewhere in the scene, the user must apply an appropriate transformation when specifying the sphere in the input file. It takes both the object-to-world and world-to-object transformations as parameters to the constructor, passing them along to the parent `Shape` constructor. Its implementation is in the files `shapes/sphere.h` and `shapes/sphere.cpp`.

The radius of the sphere can have an arbitrary positive value, and the sphere's extent can be truncated in two different ways. First, minimum and maximum z values may be set; the parts of the sphere below and above these, respectively, are cut off. Second, considering the parameterization of the sphere in spherical coordinates, a maximum ϕ value can be set. The sphere sweeps out ϕ values from 0 to the given $\phi_{\max}$ such that the section of the sphere with spherical ϕ values above $\phi_{\max}$ is also removed.

Shape 108
Sphere 115

⟨*Sphere Method Definitions*⟩ ≡
```
Sphere::Sphere(const Transform *o2w, const Transform *w2o, bool ro,
               float rad, float z0, float z1, float pm)
    : Shape(o2w, w2o, ro) {
    radius = rad;
    zmin = Clamp(min(z0, z1), -radius, radius);
    zmax = Clamp(max(z0, z1), -radius, radius);
    thetaMin = acosf(Clamp(zmin/radius, -1.f, 1.f));
    thetaMax = acosf(Clamp(zmax/radius, -1.f, 1.f));
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

⟨*Sphere Private Data*⟩ ≡ 115
```
float radius;
float phiMax;
float zmin, zmax;
float thetaMin, thetaMax;
```

3.2.2 BOUNDING

Computing a bounding box for a sphere is straightforward. The implementation here uses the values of $z_{\min}$ and $z_{\max}$ provided by the user to tighten up the bound when less than an entire sphere is being rendered. However, it doesn't do the extra work to look at $\phi_{\max}$ and see if it can compute a tighter bounding box when $\phi_{\max}$ is less than 2π. This improvement is left as an exercise.

⟨*Sphere Method Definitions*⟩ +≡
```
BBox Sphere::ObjectBound() const {
    return BBox(Point(-radius, -radius, zmin),
                Point( radius,  radius, zmax));
}
```

3.2.3 INTERSECTION

The task of deriving an intersection test is simplified by the fact that the sphere is centered at the origin. However, if the sphere has been transformed to another position in world space, then it is necessary to transform rays to object space before intersecting them with the sphere, using the world-to-object transformation. Given a ray in object space, the intersection computation can be performed in object space instead.[2]

The following fragment shows the entire intersection method:

BBox 70
Clamp() 1000
Point 63
Radians() 1001
Shape 108
Sphere 115
Sphere::phiMax 116
Sphere::radius 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Sphere::zmax 116
Sphere::zmin 116
Transform 76

2 This is something of a classic theme in computer graphics. By transforming the problem to a particular restricted case, it is possible to more easily and efficiently do an intersection test: that is, many terms of the equations cancel out since the sphere is always at $(0, 0, 0)$. No overall generality is lost, since an appropriate translation can be applied to the ray for spheres at other positions.

⟨*Sphere Method Definitions*⟩ +≡
```
bool Sphere::Intersect(const Ray &r, float *tHit, float *rayEpsilon,
                       DifferentialGeometry *dg) const {
    float phi;
    Point phit;
    ⟨Transform Ray to object space 117⟩
    ⟨Compute quadratic sphere coefficients 118⟩
    ⟨Solve quadratic equation for t values 118⟩
    ⟨Compute sphere hit position and φ 119⟩
    ⟨Test sphere intersection against clipping parameters 120⟩
    ⟨Find parametric representation of sphere hit 120⟩
    ⟨Initialize DifferentialGeometry from parametric information 122⟩
    ⟨Update tHit for quadric intersection 123⟩
    ⟨Compute rayEpsilon for quadric intersection 123⟩
    return true;
}
```

First, the given world space ray is transformed to the sphere's object space. The remainder of the intersection test will take place in that coordinate system.

⟨*Transform* Ray *to object space*⟩ ≡ 117, 123, 127, 131
```
Ray ray;
(*WorldToObject)(r, &ray);
```

If a sphere is centered at the origin with radius r, its implicit representation is

$$x^2 + y^2 + z^2 - r^2 = 0.$$

By substituting the parametric representation of the ray, given in Equation (2.3), into the implicit sphere equation, we have

$$(\mathrm{o}_x + t\mathrm{d}_x)^2 + (\mathrm{o}_y + t\mathrm{d}_y)^2 + (\mathrm{o}_z + t\mathrm{d}_z)^2 = r^2.$$

Note that all elements of this equation besides t are known values. The t values where the equation holds give the parametric positions along the ray where the implicit sphere equation holds and thus the points along the ray where it intersects the sphere. We can expand this equation and gather the coefficients for a general quadratic equation in t,

$$At^2 + Bt + C = 0,$$

where[3]

$$A = \mathrm{d}_x^2 + \mathrm{d}_y^2 + \mathrm{d}_z^2$$
$$B = 2(\mathrm{d}_x\mathrm{o}_x + \mathrm{d}_y\mathrm{o}_y + \mathrm{d}_z\mathrm{o}_z)$$
$$C = \mathrm{o}_x^2 + \mathrm{o}_y^2 + \mathrm{o}_z^2 - r^2.$$

DifferentialGeometry 102
Point 63
Ray 66
Shape::WorldToObject 108
Sphere 115

3 Some ray tracers require that the direction vector of a ray be normalized, meaning $A = 1$. This can lead to subtle errors, however, if the caller forgets to normalize the ray direction. Of course, these errors can be avoided by normalizing the direction in the ray constructor, but this wastes effort when the provided direction is *already* normalized. To avoid this needless complexity, pbrt never insists on vector normalization in intersection routines. This is particularly helpful since it reduces the amount of computation needed to transform rays to object space, because no normalization is necessary there.

This directly translates to this fragment of source code:

⟨*Compute quadratic sphere coefficients*⟩ ≡ 117, 123
```
float A = ray.d.x*ray.d.x + ray.d.y*ray.d.y + ray.d.z*ray.d.z;
float B = 2 * (ray.d.x*ray.o.x + ray.d.y*ray.o.y + ray.d.z*ray.o.z);
float C = ray.o.x*ray.o.x + ray.o.y*ray.o.y +
          ray.o.z*ray.o.z - radius*radius;
```

There are two possible solutions to this quadratic equation, giving zero, one, or two nonimaginary t values where the ray intersects the sphere:

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$
$$t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}.$$

The `Quadratic()` utility function solves a quadratic equation, returning `false` if there are no real solutions and returning `true` and setting `t0` and `t1` appropriately if there are solutions:

⟨*Solve quadratic equation for* `t` *values*⟩ ≡ 117, 123, 127
```
float t0, t1;
if (!Quadratic(A, B, C, &t0, &t1))
    return false;
⟨Compute intersection distance along ray 119⟩
```

⟨*Global Inline Functions*⟩ ≡
```
inline bool Quadratic(float A, float B, float C, float *t0, float *t1) {
    ⟨Find quadratic discriminant 118⟩
    ⟨Compute quadratic t values 119⟩
}
```

If the discriminant ($B^2 - 4AC$) is negative, then there are no real roots and the ray must miss the sphere:

⟨*Find quadratic discriminant*⟩ ≡ 118
```
float discrim = B * B - 4.f * A * C;
if (discrim <= 0.) return false;
float rootDiscrim = sqrtf(discrim);
```

The usual version of the quadratic equation can give poor numeric precision when $B \approx \pm\sqrt{B^2 - 4AC}$ due to cancellation error. It can be rewritten algebraically to a more stable form:

$$t_0 = \frac{q}{A}$$
$$t_1 = \frac{C}{q},$$

where

$$q = \begin{cases} -.5(B - \sqrt{B^2 - 4AC}) & B < 0 \\ -.5(B + \sqrt{B^2 - 4AC}) & \text{otherwise.} \end{cases}$$

Quadratic() 118
Sphere::radius 116

⟨*Compute quadratic* t *values*⟩ ≡ 118
```
float q;
if (B < 0) q = -.5f * (B - rootDiscrim);
else       q = -.5f * (B + rootDiscrim);
*t0 = q / A;
*t1 = C / q;
if (*t0 > *t1) swap(*t0, *t1);
return true;
```

Given the two intersection t values, the intersection method checks them against the ray segment from `mint` to `maxt`. Since t_0 is guaranteed to be less than t_1 (and `mint` less than `maxt`), if t_0 is greater than `maxt` or t_1 is less than `mint`, then it is certain that both hits are out of the range of interest. Otherwise, t_0 is the tentative hit distance. It may be less than `mint`, however, in which case we ignore it and try t_1. If that is also out of range, we have no valid intersection. If there is an intersection, `thit` holds the distance to the hit.

⟨*Compute intersection distance along ray*⟩ ≡ 118
```
if (t0 > ray.maxt || t1 < ray.mint)
    return false;
float thit = t0;
if (t0 < ray.mint) {
    thit = t1;
    if (thit > ray.maxt) return false;
}
```

3.2.4 PARTIAL SPHERES

Given the parametric distance along the ray to the intersection with a full sphere, it is necessary to handle partial spheres with clipped z or ϕ ranges. Intersections that are in clipped areas need to be ignored. The implementation starts by computing the object space position of the intersection, `phit`, and the ϕ value for the hit point. Using the parametric representation of the sphere,

$$\frac{y}{x} = \frac{r \sin\theta \sin\phi}{r \sin\theta \cos\phi} = \tan\phi,$$

so $\phi = \arctan y/x$. It is necessary to remap the result of the C standard library's `atan2f` function to a value between 0 and 2π, to match the sphere's original definition.

⟨*Compute sphere hit position and* ϕ⟩ ≡ 117, 120, 123
```
phit = ray(thit);
if (phit.x == 0.f && phit.y == 0.f) phit.x = 1e-5f * radius;
phi = atan2f(phit.y, phit.x);
if (phi < 0.) phi += 2.f*M_PI;
```

The hit point can now be tested against the specified minima and maxima for z and ϕ. One subtlety is that it's important to skip the z tests if the z range includes the entire sphere; the computed `phit.z` value may be slightly out of the z range due to floating-point roundoff, so we should only perform this test when the user expects the sphere to be partially incomplete. If the t_0 intersection wasn't actually valid, the routine tries again with t_1.

M_PI 1002

⟨*Test sphere intersection against clipping parameters*⟩ ≡ 117, 123
```
if ((zmin > -radius && phit.z < zmin) ||
    (zmax <  radius && phit.z > zmax) || phi > phiMax) {
    if (thit == t1) return false;
    if (t1 > ray.maxt) return false;
    thit = t1;
    ⟨Compute sphere hit position and φ 119⟩
    if ((zmin > -radius && phit.z < zmin) ||
        (zmax <  radius && phit.z > zmax) || phi > phiMax)
        return false;
}
```

At this point in the routine, it is certain that the ray hits the sphere, and the `DifferentialGeometry` structure can be initialized. The method computes u and v values by scaling the previously computed ϕ value for the hit to lie between 0 and 1 and by computing a θ value between 0 and 1 for the hit point, based on the range of θ values for the given sphere. Then, it finds the parametric partial derivatives of position $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ and surface normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$.

⟨*Find parametric representation of sphere hit*⟩ ≡ 117
```
float u = phi / phiMax;
float theta = acosf(Clamp(phit.z / radius, -1.f, 1.f));
float v = (theta - thetaMin) / (thetaMax - thetaMin);
⟨Compute sphere ∂p/∂u and ∂p/∂v 121⟩
⟨Compute sphere ∂n/∂u and ∂n/∂v 122⟩
```

Computing the partial derivatives of a point on the sphere is a short exercise in algebra. Here we will show how the x component of $\partial \mathrm{p}/\partial u$, $\partial \mathrm{p}_x/\partial u$, is calculated; the other components are found similarly. Using the parametric definition of the sphere, we have

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ \frac{\partial \mathrm{p}_x}{\partial u} &= \frac{\partial}{\partial u}(r \sin\theta \cos\phi) \\ &= r \sin\theta \frac{\partial}{\partial u}(\cos\phi) \\ &= r \sin\theta(-\phi_{\max} \sin\phi). \end{aligned}$$

Using a substitution based on the parametric definition of the sphere's y coordinate, this simplifies to

$$\frac{\partial \mathrm{p}_x}{\partial u} = -\phi_{\max} y.$$

Similarly,

$$\frac{\partial \mathrm{p}_y}{\partial u} = \phi_{\max} x,$$

and

$$\frac{\partial \mathrm{p}_z}{\partial u} = 0.$$

Clamp() 1000
DifferentialGeometry 102
Sphere::phiMax 116
Sphere::radius 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Sphere::zmax 116
Sphere::zmin 116

A similar process gives $\partial \mathrm{p}/\partial v$. The complete result is

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y, \phi_{\max} x, 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = (\theta_{\max} - \theta_{\min})(z \cos\phi, z \sin\phi, -r \sin\theta).$$

⟨*Compute sphere* $\partial \mathrm{p}/\partial u$ *and* $\partial \mathrm{p}/\partial v$⟩ ≡ 120

```
float zradius = sqrtf(phit.x*phit.x + phit.y*phit.y);
float invzradius = 1.f / zradius;
float cosphi = phit.x * invzradius;
float sinphi = phit.y * invzradius;
Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0);
Vector dpdv = (thetaMax-thetaMin) *
    Vector(phit.z * cosphi, phit.z * sinphi,
           -radius * sinf(theta));
```

★ 3.2.5 PARTIAL DERIVATIVES OF NORMAL VECTORS

It is also useful to determine how the normal changes as we move along the surface in the u and v directions. For example, the antialiasing techniques in Chapter 10 are dependent on this information to antialias textures on objects that are seen reflected in curved surfaces. The differential changes in normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ are given by the *Weingarten equations* from differential geometry:

$$\frac{\partial \mathbf{n}}{\partial u} = \frac{fF - eG}{EG - F^2}\frac{\partial \mathrm{p}}{\partial u} + \frac{eF - fE}{EG - F^2}\frac{\partial \mathrm{p}}{\partial v}$$
$$\frac{\partial \mathbf{n}}{\partial v} = \frac{gF - fG}{EG - F^2}\frac{\partial \mathrm{p}}{\partial u} + \frac{fF - gE}{EG - F^2}\frac{\partial \mathrm{p}}{\partial v},$$

where E, F, and G are coefficients of the *first fundamental form* and are given by

$$E = \left|\frac{\partial \mathrm{p}}{\partial u}\right|^2$$
$$F = \left(\frac{\partial \mathrm{p}}{\partial u} \cdot \frac{\partial \mathrm{p}}{\partial v}\right)$$
$$G = \left|\frac{\partial \mathrm{p}}{\partial v}\right|^2.$$

These are easily computed with the $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ values found earlier. The e, f, and g are coefficients of the *second fundamental form*,

$$e = \left(\mathbf{n} \cdot \frac{\partial^2 \mathrm{p}}{\partial u^2}\right)$$
$$f = \left(\mathbf{n} \cdot \frac{\partial^2 \mathrm{p}}{\partial u \partial v}\right)$$
$$g = \left(\mathbf{n} \cdot \frac{\partial^2 \mathrm{p}}{\partial v^2}\right).$$

Sphere::phiMax 116
Sphere::radius 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Vector 57

The two fundamental forms have basic connections with the local curvature of a surface; see a differential geometry textbook such as Gray (1993) for details. To find e, f, and g, it is necessary to compute the second-order partial derivatives $\partial^2\mathrm{p}/\partial u^2$ and so on.

For spheres, a little more algebra gives the second derivatives:

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2(x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = (\theta_{\max} - \theta_{\min})\, z\, \phi_{\max}(-\sin\phi, \cos\phi, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = -(\theta_{\max} - \theta_{\min})^2(x, y, z).$$

⟨*Compute sphere* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$⟩ ≡ 120
```
Vector d2Pduu = -phiMax * phiMax * Vector(phit.x, phit.y, 0);
Vector d2Pduv = (thetaMax - thetaMin) * phit.z * phiMax *
                Vector(-sinphi, cosphi, 0.);
Vector d2Pdvv = -(thetaMax - thetaMin) * (thetaMax - thetaMin) *
                Vector(phit.x, phit.y, phit.z);
⟨Compute coefficients for fundamental forms 122⟩
⟨Compute ∂n/∂u and ∂n/∂v from fundamental form coefficients 122⟩
```

⟨*Compute coefficients for fundamental forms*⟩ ≡ 122, 128
```
float E = Dot(dpdu, dpdu);
float F = Dot(dpdu, dpdv);
float G = Dot(dpdv, dpdv);
Vector N = Normalize(Cross(dpdu, dpdv));
float e = Dot(N, d2Pduu);
float f = Dot(N, d2Pduv);
float g = Dot(N, d2Pdvv);
```

⟨*Compute* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$ *from fundamental form coefficients*⟩ ≡ 122, 128
```
float invEGF2 = 1.f / (E*G - F*F);
Normal dndu = Normal((f*F - e*G) * invEGF2 * dpdu +
                     (e*F - f*E) * invEGF2 * dpdv);
Normal dndv = Normal((g*F - f*G) * invEGF2 * dpdu +
                     (f*F - g*E) * invEGF2 * dpdv);
```

3.2.6 DifferentialGeometry INITIALIZATION

Having computed the surface parameterization and all the relevant partial derivatives, the DifferentialGeometry structure can be initialized with the geometric information for this intersection:

⟨*Initialize* DifferentialGeometry *from parametric information*⟩ ≡ 117, 127, 131
```
const Transform &o2w = *ObjectToWorld;
*dg = DifferentialGeometry(o2w(phit), o2w(dpdu), o2w(dpdv),
                           o2w(dndu), o2w(dndv), u, v, this);
```

DifferentialGeometry 102
Dot() 60
Normal 65
Shape::ObjectToWorld 108
Sphere::phiMax 116
Sphere::thetaMax 116
Sphere::thetaMin 116
Sphere::zmax 116
Sphere::zmin 116
Transform 76
Vector 57

Since there is an intersection, the `tHit` parameter is updated with the parametric hit distance along the ray, which was stored in `thit`. This will allow subsequent intersection tests to terminate early if the potential hit would be farther away than the existing intersection.

A natural question to ask at this point is, "What effect does the world-to-object transformation have on the correct parametric distance to return?" Indeed, the intersection method has found a parametric distance to the intersection for the object space ray, which may have been translated, rotated, scaled, or worse when it was transformed from world space. However, it can be shown that the parametric distance to an intersection in object space is exactly the same as it would have been if the ray was left in world space and the intersection had been done there and, thus, `tHit` can be initialized directly. Note that if the object space ray's direction had been normalized after the transformation, then this would no longer be the case and a correction factor related to the unnormalized ray's length would be needed. This is another motivation for not normalizing the object space ray's direction vector after transformation.

⟨*Update* `tHit` *for quadric intersection*⟩ ≡ **117, 127, 131**
```
*tHit = thit;
```

We have not rigorously derived floating-point error bounds for the sphere ray–object intersection algorithm implemented here, leaving that for an exercise at the end of the chapter. However, we have found that this expression generally works well.

⟨*Compute* `rayEpsilon` *for quadric intersection*⟩ ≡ **117, 127, 131**
```
*rayEpsilon = 5e-4f * *tHit;
```

The `Sphere::IntersectP()` routine is almost identical to `Sphere::Intersect()`, but it does not fill in the `DifferentialGeometry` structure. Because the `Intersect()` and `IntersectP()` methods are always so closely related, we will not show implementations of `IntersectP()` for the remaining shapes.

⟨*Sphere Method Definitions*⟩ +≡
```
bool Sphere::IntersectP(const Ray &r) const {
    float phi;
    Point phit;
    ⟨Transform Ray to object space 117⟩
    ⟨Compute quadratic sphere coefficients 118⟩
    ⟨Solve quadratic equation for t values 118⟩
    ⟨Compute sphere hit position and φ 119⟩
    ⟨Test sphere intersection against clipping parameters 120⟩
    return true;
}
```

3.2.7 SURFACE AREA

Point 63
Ray 66
Sphere 115
Sphere::Intersect() 117

To compute the surface area of quadrics, we use a standard formula from integral calculus. If a curve $y = f(x)$ from $y = a$ to $y = b$ is revolved around the x axis, the surface area of the resulting swept surface is

$$2\pi \int_a^b f(x)\sqrt{1 + \left(f'(x)\right)^2}\,\mathrm{d}x,$$

where $f'(x)$ denotes the derivative $\mathrm{d}f/\mathrm{d}x$.[4] Since most of our surfaces of revolution are only partially swept around the axis, we will instead use the formula

$$\phi_{\max} \int_a^b f(x)\sqrt{1 + \left(f'(x)\right)^2}\,\mathrm{d}x.$$

The sphere is a surface of revolution of a circular arc. The function that defines the profile curve of the sphere is

$$f(x) = \sqrt{r^2 - x^2},$$

and its derivative is

$$f'(x) = -\frac{x}{\sqrt{r^2 - x^2}}.$$

Recall that the sphere is clipped at $z_{\min}$ and $z_{\max}$. The surface area is therefore

$$\begin{aligned} A &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} \sqrt{r^2 - x^2}\sqrt{1 + \frac{x^2}{r^2 - x^2}}\,\mathrm{d}x \\ &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} \sqrt{r^2 - x^2 + x^2}\,\mathrm{d}x \\ &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} r\,\mathrm{d}x \\ &= \phi_{\max}\, r\, (z_{\max} - z_{\min}). \end{aligned}$$

For the full sphere $\phi_{\max} = 2\pi$, $z_{\min} = -r$, and $z_{\max} = r$, so we have the standard formula $A = 4\pi r^2$, confirming that the formula makes sense.

⟨*Sphere Method Definitions*⟩ +≡

```
float Sphere::Area() const {
    return phiMax * radius * (zmax-zmin);
}
```

3.3 CYLINDERS

⟨*Cylinder Declarations*⟩ ≡

```
class Cylinder : public Shape {
public:
    ⟨Cylinder Public Methods⟩
protected:
    ⟨Cylinder Private Data 126⟩
};
```

Shape 108
Sphere 115
Sphere::phiMax 116
Sphere::radius 116
Sphere::zmax 116
Sphere::zmin 116

4 See Anton, Bivens, and Davis (2001) for a derivation.

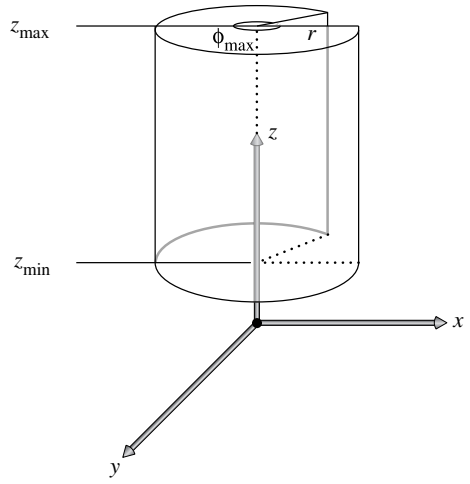


Figure 3.5: Basic Setting for the Cylinder Shape. It has a radius of r and covers a range along the z axis. A partial cylinder may be swept by specifying a maximum ϕ value.

3.3.1 CONSTRUCTION

Another useful quadric is the cylinder; `pbrt` provides cylinder `Shapes` that are centered around the z axis. The implementation is in the files `shapes/cylinder.h` and `shapes/cylinder.cpp`. The user supplies a minimum and maximum z value for the cylinder, as well as a radius and maximum ϕ sweep value (Figure 3.5). In parametric form, a cylinder is described by the following equations:

$$\begin{aligned} \phi &= u\,\phi_{\max} \\ x &= r\cos\phi \\ y &= r\sin\phi \\ z &= z_{\min} + v(z_{\max} - z_{\min}). \end{aligned}$$

Figure 3.6 shows a rendered image of two cylinders. Like the sphere image, the left cylinder is a complete cylinder, while the right one is a partial cylinder due to having a $\phi_{\max}$ value less than 2π.

Clamp() 1000
Cylinder 124
Cylinder::phiMax 126
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Radians() 1001
Shape 108
Transform 76

⟨*Cylinder Method Definitions*⟩ ≡

```
Cylinder::Cylinder(const Transform *o2w, const Transform *w2o, bool ro,
                   float rad, float z0, float z1, float pm)
    : Shape(o2w, w2o, ro) {
    radius = rad;
    zmin = min(z0, z1);
    zmax = max(z0, z1);
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

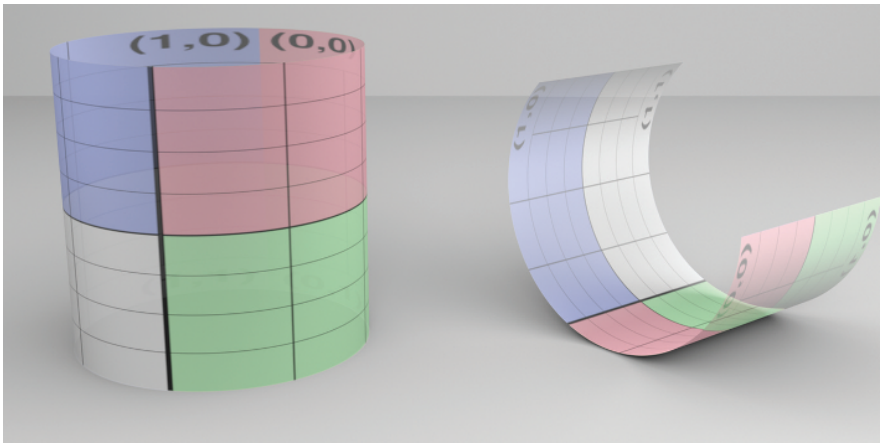


Figure 3.6: Two Cylinders. A complete cylinder is on the left, and a partial cylinder is on the right.

⟨*Cylinder Private Data*⟩ ≡ 124
```
float radius, zmin, zmax, phiMax;
```

3.3.2 BOUNDING

As was done with the sphere, the bounding method computes a conservative bounding box for the cylinder using the z range but without taking into account the maximum ϕ.

⟨*Cylinder Method Definitions*⟩ +≡
```
BBox Cylinder::ObjectBound() const {
    Point p1 = Point(-radius, -radius, zmin);
    Point p2 = Point( radius,  radius, zmax);
    return BBox(p1, p2);
}
```

3.3.3 INTERSECTION

The ray–cylinder intersection formula can be found by substituting the ray equation into the cylinder's implicit equation, similarly to the sphere case. The implicit equation for an infinitely long cylinder centered on the z axis with radius r is

$$x^2 + y^2 - r^2 = 0.$$

Substituting the ray equation, Equation (2.3), we have

$$(o_x + td_x)^2 + (o_y + td_y)^2 = r^2.$$

When we expand this and find the coefficients of the quadratic equation $At^2 + Bt + C$, we have

BBox 70
Cylinder 124
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Point 63

$$A = \mathbf{d}_x^2 + \mathbf{d}_y^2$$
$$B = 2(\mathbf{d}_x \mathbf{o}_x + \mathbf{d}_y \mathbf{o}_y)$$
$$C = \mathbf{o}_x^2 + \mathbf{o}_y^2 - r^2.$$

⟨*Compute quadratic cylinder coefficients*⟩ ≡ 127
```
float A = ray.d.x*ray.d.x + ray.d.y*ray.d.y;
float B = 2 * (ray.d.x*ray.o.x + ray.d.y*ray.o.y);
float C = ray.o.x*ray.o.x + ray.o.y*ray.o.y - radius*radius;
```

The solution process for the quadratic equation is similar for all quadric shapes, so some fragments from the `Sphere` intersection method will be reused in the following.

⟨*Cylinder Method Definitions*⟩ +≡
```
bool Cylinder::Intersect(const Ray &r, float *tHit, float *rayEpsilon,
                         DifferentialGeometry *dg) const {
    float phi;
    Point phit;
    ⟨Transform Ray to object space 117⟩
    ⟨Compute quadratic cylinder coefficients 127⟩
    ⟨Solve quadratic equation for t values 118⟩
    ⟨Compute cylinder hit point and φ 127⟩
    ⟨Test cylinder intersection against clipping parameters 128⟩
    ⟨Find parametric representation of cylinder hit 128⟩
    ⟨Initialize DifferentialGeometry from parametric information 122⟩
    ⟨Update tHit for quadric intersection 123⟩
    ⟨Compute rayEpsilon for quadric intersection 123⟩
    return true;
}
```

3.3.4 PARTIAL CYLINDERS

As with the sphere, we can invert the parametric description of the cylinder to compute a ϕ value by inverting the x and y parametric equations to solve for ϕ. In fact, the result is the same as for the sphere.

⟨*Compute cylinder hit point and φ*⟩ ≡ 127, 128
```
phit = ray(thit);
phi = atan2f(phit.y, phit.x);
if (phi < 0.) phi += 2.f*M_PI;
```

Cylinder 124
DifferentialGeometry 102
M_PI 1002
Point 63
Ray 66
Sphere 115

The intersection routine now makes sure that the hit is in the specified z range, and that the angle ϕ is acceptable. If not, it rejects the hit and tries with t_1, if it hasn't already, just like the sphere.

⟨*Test cylinder intersection against clipping parameters*⟩ ≡ 127
```
if (phit.z < zmin || phit.z > zmax || phi > phiMax) {
    if (thit == t1) return false;
    thit = t1;
    if (t1 > ray.maxt) return false;
    ⟨Compute cylinder hit point and φ 127⟩
    if (phit.z < zmin || phit.z > zmax || phi > phiMax)
        return false;
}
```

Again the u value is computed by scaling ϕ to lie between 0 and 1. Straightforward inversion of the parametric equation for the cylinder's z value gives the v parametric coordinate.

⟨*Find parametric representation of cylinder hit*⟩ ≡ 127
```
float u = phi / phiMax;
float v = (phit.z - zmin) / (zmax - zmin);
⟨Compute cylinder ∂p/∂u and ∂p/∂v 128⟩
⟨Compute cylinder ∂n/∂u and ∂n/∂v 128⟩
```

The partial derivatives for a cylinder are quite easy to derive:

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y,\ \phi_{\max} x,\ 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = (0,\ 0,\ z_{\max} - z_{\min}).$$

⟨*Compute cylinder ∂p/∂u and ∂p/∂v*⟩ ≡ 128
```
Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0);
Vector dpdv(0, 0, zmax - zmin);
```

We again use the Weingarten equations to compute the parametric change in the cylinder normal. The relevant partial derivatives are

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2 (x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = (0, 0, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = (0, 0, 0).$$

⟨*Compute cylinder ∂n/∂u and ∂n/∂v*⟩ ≡ 128
```
Vector d2Pduu = -phiMax * phiMax * Vector(phit.x, phit.y, 0);
Vector d2Pduv(0, 0, 0), d2Pdvv(0, 0, 0);
⟨Compute coefficients for fundamental forms 122⟩
⟨Compute ∂n/∂u and ∂n/∂v from fundamental form coefficients 122⟩
```

Cylinder::phiMax 126
Cylinder::zmax 126
Cylinder::zmin 126
Vector 57

3.3.5 SURFACE AREA

A cylinder is just a rolled-up rectangle. If you unroll the rectangle, its height is $z_{\max} - z_{\min}$, and its width is $r\phi_{\max}$:

⟨*Cylinder Method Definitions*⟩ +≡
```
float Cylinder::Area() const {
    return (zmax-zmin) * phiMax * radius;
}
```

3.4 DISKS

⟨*Disk Declarations*⟩ ≡
```
class Disk : public Shape {
public:
    ⟨Disk Public Methods⟩
private:
    ⟨Disk Private Data 130⟩
};
```

The disk is an interesting quadric since it has a particularly straightforward intersection routine that avoids solving the quadratic equation. In `pbrt`, a `Disk` is a circular disk of radius r at height h along the z axis. It is implemented in the files `shapes/disk.h` and `shapes/disk.cpp`. In order to make partial disks, the user may specify a maximum ϕ value beyond which the disk is cut off (Figure 3.7). The disk can also be generalized to an

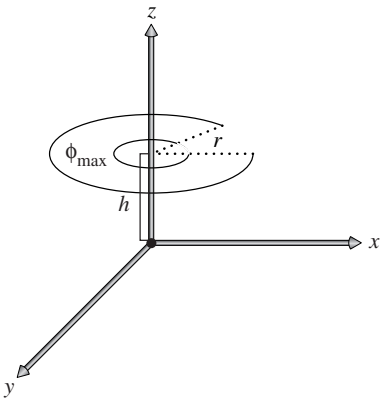


Figure 3.7: Basic Setting for the Disk Shape. The disk has radius r and is located at height h along the z axis. A partial disk may be swept by specifying a maximum ϕ value and an inner radius r_i.

Cylinder 124
Cylinder::phiMax 126
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Disk 129
Shape 108

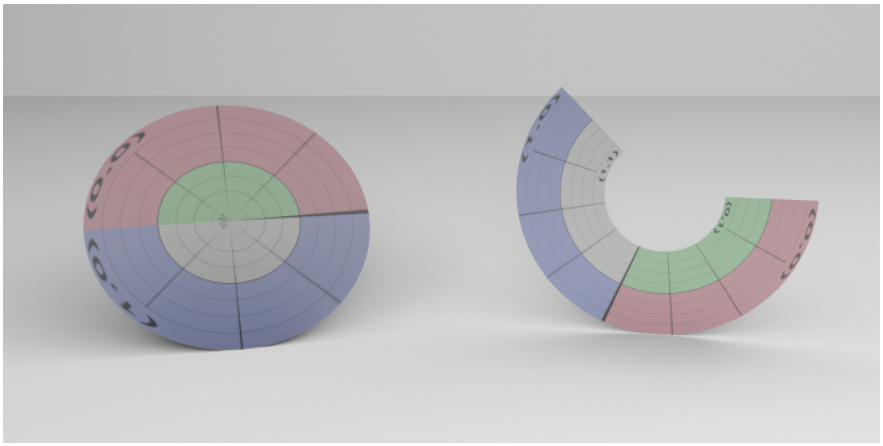


Figure 3.8: Two Disks. A complete disk is on the left, and a partial disk is on the right.

annulus by specifying an inner radius, r_i. In parametric form, it is described by

$$\begin{aligned} \phi &= u\,\phi_{\max} \\ x &= ((1-v)r_i + vr)\cos\phi \\ y &= ((1-v)r_i + vr)\sin\phi \\ z &= h. \end{aligned}$$

Figure 3.8 is a rendered image of two disks.

3.4.1 CONSTRUCTION

⟨*Disk Method Definitions*⟩ ≡

```
Disk::Disk(const Transform *o2w, const Transform *w2o, bool ro,
           float ht, float r, float ri, float tmax)
    : Shape(o2w, w2o, ro) {
    height = ht;
    radius = r;
    innerRadius = ri;
    phiMax = Radians(Clamp(tmax, 0.0f, 360.0f));
}
```

⟨*Disk Private Data*⟩ ≡ 129

```
float height, radius, innerRadius, phiMax;
```

Clamp() 1000
Disk 129
Disk::phiMax 130
Radians() 1001
Shape 108
Transform 76

3.4.2 BOUNDING

The bounding method is quite straightforward; it computes a bounding box centered at the height of the disk along z, with extent of `radius` in both the x and y directions.

⟨*Disk Method Definitions*⟩ +≡
```
BBox Disk::ObjectBound() const {
    return BBox(Point(-radius, -radius, height),
                Point( radius,  radius, height));
}
```

3.4.3 INTERSECTION

Intersecting a ray with a disk is also quite easy. The intersection of the ray with the $z = h$ plane that the disk lies in is found and the intersection point is checked to see if it lies inside the disk.

⟨*Disk Method Definitions*⟩ +≡
```
bool Disk::Intersect(const Ray &r, float *tHit, float *rayEpsilon,
                     DifferentialGeometry *dg) const {
    ⟨Transform Ray to object space 117⟩
    ⟨Compute plane intersection for disk 131⟩
    ⟨See if hit point is inside disk radii and φmax 132⟩
    ⟨Find parametric representation of disk hit 132⟩
    ⟨Initialize DifferentialGeometry from parametric information 122⟩
    ⟨Update tHit for quadric intersection 123⟩
    ⟨Compute rayEpsilon for quadric intersection 123⟩
    return true;
}
```

The first step is to compute the parametric t value where the ray intersects the plane that the disk lies in. We want to find t such that the z component of the ray's position is equal to the height of the disk. Thus,

$$h = \mathrm{o}_z + t\mathbf{d}_z$$

and

$$t = \frac{h - \mathrm{o}_z}{\mathbf{d}_z}.$$

If the ray is parallel to the disk's plane (i.e., the z component of its direction is zero), there is no intersection. Otherwise, the intersection method then checks if t is inside the legal range of values [`mint`, `maxt`]. If not, the routine can return `false`.

⟨*Compute plane intersection for disk*⟩ ≡ 131
```
if (fabsf(ray.d.z) < 1e-7) return false;
float thit = (height - ray.o.z) / ray.d.z;
if (thit < ray.mint || thit > ray.maxt)
    return false;
```

Now the intersection method can compute the point `phit` where the ray intersects the plane. Once the plane intersection is known, `false` is returned if the distance from the hit to the center of the disk is more than `Disk::radius` or less than `Disk::innerRadius`. This process can be optimized by actually computing the squared distance to the center,

BBox 70
DifferentialGeometry 102
Disk 129
Disk::height 130
Disk::innerRadius 130
Disk::radius 130
Point 63
Ray 66

taking advantage of the fact that the x and y coordinates of the center point (0, 0, `height`) are zero, and the z coordinate of `phit` is equal to `height`.

⟨*See if hit point is inside disk radii and $\phi_{\max}$*⟩ ≡ 131
```
Point phit = ray(thit);
float dist2 = phit.x * phit.x + phit.y * phit.y;
if (dist2 > radius * radius || dist2 < innerRadius * innerRadius)
    return false;
⟨Test disk φ value against φmax 132⟩
```

If the distance check passes, a final test makes sure that the ϕ value of the hit point is between zero and $\phi_{\max}$, specified by the caller. Inverting the disk's parameterization gives the same expression for ϕ as the other quadric shapes.

⟨*Test disk ϕ value against $\phi_{\max}$*⟩ ≡ 132
```
float phi = atan2f(phit.y, phit.x);
if (phi < 0) phi += 2. * M_PI;
if (phi > phiMax)
    return false;
```

If we've gotten this far, there is an intersection with the disk. The parameter `u` is scaled to reflect the partial disk specified by $\phi_{\max}$, and `v` is computed by inverting the parametric equation. The equations for the partial derivatives at the hit point can be derived with a process similar to that used for the previous quadrics. Because the normal of a disk is the same everywhere, the partial derivatives $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ are both trivially (0, 0, 0).

⟨*Find parametric representation of disk hit*⟩ ≡ 131
```
float u = phi / phiMax;
float v = 1.f - ((sqrtf(dist2)-innerRadius) /
                 (radius-innerRadius));
Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0.);
Vector dpdv(-phit.x / (1-v), -phit.y / (1-v), 0.);
dpdu *= phiMax * INV_TWOPI;
dpdv *= (radius - innerRadius) / radius;
Normal dndu(0,0,0), dndv(0,0,0);
```

3.4.4 SURFACE AREA

Disks have trivially computed surface area, since they're just portions of an annulus:

$$A = \frac{\phi_{\max}}{2}(r^2 - r_{\mathrm{i}}^2).$$

⟨*Disk Method Definitions*⟩ +≡
```
float Disk::Area() const {
    return phiMax * 0.5f *
       (radius * radius - innerRadius * innerRadius);
}
```

Disk 129
Disk::innerRadius 130
Disk::phiMax 130
Disk::radius 130
INV_TWOPI 1002
M_PI 1002
Normal 65
Point 63
Vector 57

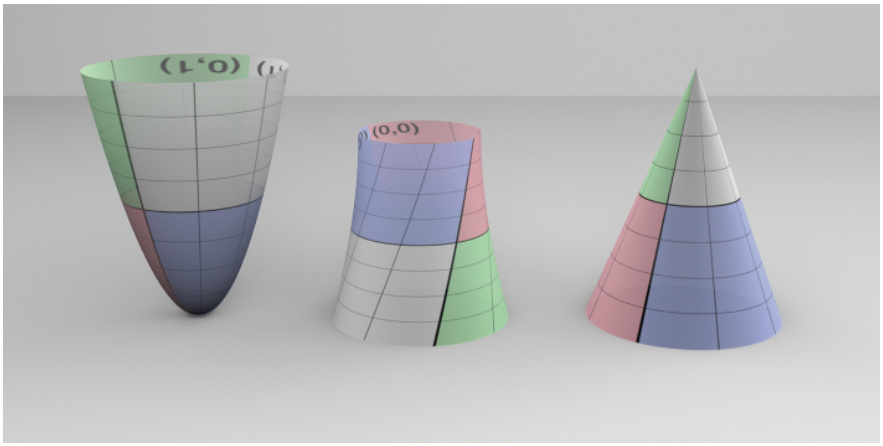


Figure 3.9: The Remaining Quadric Shapes. From left to right: the paraboloid, the hyperboloid, and the cone.

3.5 OTHER QUADRICS

`pbrt` supports three more quadrics: cones, paraboloids, and hyperboloids. They are implemented in the source files `shapes/cone.h`, `shapes/cone.cpp`, `shapes/paraboloid.h`, `shapes/paraboloid.cpp`, `shapes/hyperboloid.h`, and `shapes/hyperboloid.cpp`. We won't include their full implementations here, since the techniques used to derive their quadratic intersection coefficients, parametric coordinates, and partial derivatives should now be familiar. However, we will briefly summarize the implicit and parametric forms of these shapes. A rendered image of the three of them is in Figure 3.9.

3.5.1 CONES

The implicit equation of a cone centered on the z axis with radius r and height h is

$$\left(\frac{hx}{r}\right)^2 + \left(\frac{hy}{r}\right)^2 - (z-h)^2 = 0.$$

Cones are also described parametrically:

$$\begin{aligned} \phi &= u\,\phi_{\max} \\ x &= r(1-v)\cos\phi \\ y &= r(1-v)\sin\phi \\ z &= vh. \end{aligned}$$

The partial derivatives at a point on a cone are

$$\begin{aligned} \frac{\partial \mathrm{p}}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0) \\ \frac{\partial \mathrm{p}}{\partial v} &= \left(-\frac{x}{1-v}, \frac{y}{1-v}, h\right), \end{aligned}$$

and the second partial derivatives are

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2(x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = \frac{\phi_{\max}}{1-v}(y, -x, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = (0, 0, 0).$$

3.5.2 PARABOLOIDS

The implicit equation of a paraboloid centered on the z axis with radius r and height h is

$$\frac{hx^2}{r^2} + \frac{hy^2}{r^2} - z = 0,$$

and its parametric form is

$$\phi = u\ \phi_{\max}$$
$$z = v(z_{\max} - z_{\min})$$
$$r = r_{\max}\sqrt{\frac{z}{z_{\max}}}$$
$$x = r\ \cos\phi$$
$$y = r\ \sin\phi.$$

The partial derivatives are

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y, \phi_{\max} x, 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = (z_{\max} - z_{\min})\left(\frac{x}{2z}, \frac{y}{2z}, 1\right),$$

and

$$\frac{\partial^2 \mathrm{p}}{\partial u^2} = -\phi_{\max}^2(x, y, 0)$$
$$\frac{\partial^2 \mathrm{p}}{\partial u \partial v} = \phi_{\max}(z_{\max} - z_{\min})\left(-\frac{y}{2z}, \frac{x}{2z}, 0\right)$$
$$\frac{\partial^2 \mathrm{p}}{\partial v^2} = -(z_{\max} - z_{\min})^2\left(\frac{x}{4z^2}, \frac{y}{4z^2}, 0\right).$$

3.5.3 HYPERBOLOIDS

Finally, the implicit form of the hyperboloid is

$$x^2 + y^2 - z^2 = -1,$$

and the parametric form is

$$\begin{aligned}
\phi &= u\,\phi_{\max} \\
x_r &= (1-v)x_1 + v\,x_2 \\
y_r &= (1-v)y_1 + v\,y_2 \\
x &= x_r \cos\phi - y_r \sin\phi \\
y &= x_r \sin\phi + y_r \cos\phi \\
z &= (1-v)z_1 + v\,z_2.
\end{aligned}$$

The partial derivatives are

$$\frac{\partial \mathrm{p}}{\partial u} = (-\phi_{\max} y, \phi_{\max} x, 0)$$
$$\frac{\partial \mathrm{p}}{\partial v} = ((x_2 - x_1)\cos\phi - (y_2 - y_1)\sin\phi, (x_2 - x_1)\sin\phi + (y_2 - y_1)\cos\phi, z_2 - z_1),$$

and

$$\begin{aligned}
\frac{\partial^2 \mathrm{p}}{\partial u^2} &= -\phi_{\max}^2 (x, y, 0) \\
\frac{\partial^2 \mathrm{p}}{\partial u \partial v} &= \phi_{\max} \left(-\frac{\partial \mathrm{p}_y}{\partial v}, \frac{\partial \mathrm{p}_x}{\partial v}, 0 \right) \\
\frac{\partial^2 \mathrm{p}}{\partial v^2} &= (0, 0, 0).
\end{aligned}$$

3.6 TRIANGLES AND MESHES

⟨*TriangleMesh Declarations*⟩ ≡
```
class TriangleMesh : public Shape {
public:
    ⟨TriangleMesh Public Methods 138⟩
protected:
    ⟨TriangleMesh Protected Data 137⟩
};
```

The triangle is one of the most commonly used shapes in computer graphics. `pbrt` supports *triangle meshes*, where a number of triangles are stored together so that their per-vertex data (position, shading normal, etc.) can be shared among multiple triangles. Single triangles are simply treated as degenerate meshes. Figure 3.10 shows an image of a complex triangle mesh of over one million triangles.

The arguments to the `TriangleMesh` constructor are as follows:

- `nt`: Number of triangles.
- `nv`: Number of vertices.
- `vi`: Pointer to an array of vertex indices. For the `i`th triangle, its three vertex positions are `P[vi[3*i]]`, `P[vi[3*i+1]]`, and `P[vi[3*i+2]]`.
- `P`: Array of `nv` vertex positions.
- `N`: An optional array of normal vectors, one per vertex in the mesh. If present, these are interpolated across triangle faces to compute shading differential geometry.

Shape 108
TriangleMesh 135



Figure 3.10: Happy Buddha Model. This triangle mesh contains over one million individual triangles. It was created from a real statue using a 3D laser scanner. *(Model courtesy of Stanford Computer Graphics Laboratory Scanning Repository.)*

- `S`: An optional array of tangent vectors, one per vertex in the mesh. These are also used to compute shading geometry.
- `uv`: An optional array of parametric (u, v) values, one for each vertex.
- `atex`: An optional *alpha mask* texture, which can be used to cut away part of the triangle's surface.

The constructor copies the relevant information and stores it in the `TriangleMesh` object. In particular, it must make its own copies of `vi`, `P`, `N`, and `S`, since the caller retains ownership of the data being passed in.

`TriangleMesh` 135

Triangles have a dual role among the primitives in `pbrt`: not only are they a user-supplied primitive, but other primitives often tessellate themselves into triangle meshes. For exam-

ple, subdivision surfaces end up creating a mesh of triangles to approximate the smooth limit surface. Ray intersections are performed against these triangles, rather than directly against the subdivision surface. (See Section 3.7.3.)

Due to this second role, it's important that a routine that is creating a triangle mesh be able to specify the parameterization of the triangles. If a triangle was created by evaluating the position of a parametric surface at three particular (u, v) coordinate values, for example, those (u, v) values should be interpolated to compute the (u, v) value at ray intersection points inside the triangle; hence the `uv` parameter. The (u, v) values are also useful for texture mapping, where an external program that created a triangle mesh may want to assign (u, v) coordinates to the mesh so that a texture map assigns color to the mesh surface in the desired way.

⟨*TriangleMesh Method Definitions*⟩ ≡
```
TriangleMesh::TriangleMesh(const Transform *o2w, const Transform *w2o,
        bool ro, int nt, int nv, const int *vi, const Point *P,
        const Normal *N, const Vector *S, const float *uv,
        const Reference<Texture<float> > &atex)
    : Shape(o2w, w2o, ro), alphaTexture(atex) {
    ntris = nt;
    nverts = nv;
    vertexIndex = new int[3 * ntris];
    memcpy(vertexIndex, vi, 3 * ntris * sizeof(int));
    ⟨Copy uv, N, and S vertex data, if present⟩
    ⟨Transform mesh vertices to world space 138⟩
}
```

The ⟨*Copy* `uv`, `N`, *and* `S` *vertex data, if present*⟩ fragment just allocates the appropriate amount of space and copies any data. Its implementation isn't included here.

⟨*TriangleMesh Protected Data*⟩ ≡ 135
```
int ntris, nverts;
int *vertexIndex;
Point *p;
Normal *n;
Vector *s;
float *uvs;
Reference<Texture<float> > alphaTexture;
```

Normal 65
Point 63
Reference 1011
Shape 108
Texture 519
Transform 76
TriangleMesh 135
Vector 57

Unlike the other shapes that leave the primitive description in object space and then transform incoming rays from world space to object space, triangle meshes transform the shape into world space and thus save the work of transforming incoming rays into object space and the work of transforming the intersection's differential geometry out to world space. This is a good idea because this operation can be performed once at start-up, avoiding transforming rays many times during rendering. Doing this with quadrics is more complicated, although possible—see Exercise 3.1 at the end of the chapter for more information. Normal and tangent vectors for shading geometry are left in object space, since the `GetShadingGeometry()` method must transform them to world space with the

transformation matrix supplied to that method, which may not necessarily be the one stored by the Shape.

⟨*Transform mesh vertices to world space*⟩ ≡ 137
```
for (int i = 0; i < nverts; ++i)
    p[i] = (*ObjectToWorld)(P[i]);
```

The object space bound of a triangle mesh is easily found by computing a bounding box that encompasses all of the vertices of the mesh. Because the vertex positions p are transformed to world space in the constructor, the implementation here has to transform them back to object space before computing their bound. Note that these bounds are fairly expensive to compute for large triangle meshes. The implementation assumes that the caller will cache the results of these computations if they will be reused, which is the case elsewhere in pbrt where the bounding methods are called.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox TriangleMesh::ObjectBound() const {
    BBox objectBounds;
    for (int i = 0; i < nverts; i++)
        objectBounds = Union(objectBounds, (*WorldToObject)(p[i]));
    return objectBounds;
}
```

The TriangleMesh shape is one of the shapes that can compute a better world space bound than can be found by transforming its object space bounding box to world space. Its world space bound can be directly computed from the world space vertices.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox TriangleMesh::WorldBound() const {
    BBox worldBounds;
    for (int i = 0; i < nverts; i++)
        worldBounds = Union(worldBounds, p[i]);
    return worldBounds;
}
```

The TriangleMesh shape does not directly compute intersections. Instead, it splits itself into many separate Triangles, each representing a single triangle. All of the individual triangles reference the shared set of vertices in p, avoiding per-triangle replication of the shared data. The TriangleMesh class therefore overrides the Shape::CanIntersect() method to indicate that TriangleMeshes cannot be intersected directly.

⟨*TriangleMesh Public Methods*⟩ ≡ 135
```
bool CanIntersect() const { return false; }
```

When pbrt encounters a shape that cannot be intersected directly, it calls its Refine() method. Shape::Refine() is expected to produce a list of simpler shapes in the vector passed in, refined. The implementation here is simple; it creates a new Triangle for each of the triangles in the mesh.

BBox 70
Shape::CanIntersect() 110
Shape::ObjectToWorld 108
Shape::Refine() 110
Shape::WorldToObject 108
Triangle 139
TriangleMesh 135
TriangleMesh::nverts 137
TriangleMesh::p 137
Union() 72

⟨*TriangleMesh Method Definitions*⟩ +≡
```
void TriangleMesh::Refine(vector<Reference<Shape> > &refined) const {
    for (int i = 0; i < ntris; ++i)
        refined.push_back(new Triangle(ObjectToWorld,
                          WorldToObject, ReverseOrientation,
                          (TriangleMesh *)this, i));
}
```

3.6.1 TRIANGLE

⟨*TriangleMesh Declarations*⟩ +≡
```
class Triangle : public Shape {
public:
    ⟨Triangle Public Methods 139⟩
private:
    ⟨Triangle Private Data 139⟩
};
```

The `Triangle` doesn't store much data—just a pointer to the parent `TriangleMesh` that it came from and a pointer to its three vertex indices in the mesh:

⟨*Triangle Public Methods*⟩ ≡ 139
```
Triangle(const Transform *o2w, const Transform *w2o, bool ro,
         TriangleMesh *m, int n)
    : Shape(o2w, w2o, ro) {
    mesh = m;
    v = &mesh->vertexIndex[3*n];
}
```

Note that the implementation stores a pointer to the first vertex *index*, instead of storing three pointers to the vertices themselves. This reduces the amount of storage required for each `Triangle` at a cost of another level of indirection.

⟨*Triangle Private Data*⟩ ≡ 139
```
Reference<TriangleMesh> mesh;
int *v;
```

As with `TriangleMeshes`, it is possible to compute better world space bounding boxes for individual triangles by bounding the world space vertices directly.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox Triangle::ObjectBound() const {
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    return Union(BBox((*WorldToObject)(p1), (*WorldToObject)(p2)),
                 (*WorldToObject)(p3));
}
```

BBox 70
Reference 1011
Shape 108
Shape::ObjectToWorld 108
Shape::ReverseOrientation 108
Shape::WorldToObject 108
Transform 76
Triangle 139
TriangleMesh 135
TriangleMesh::ntris 137
TriangleMesh::vertexIndex 137

⟨*TriangleMesh Method Definitions*⟩ +≡
```
BBox Triangle::WorldBound() const {
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    return Union(BBox(p1, p2), p3);
}
```

⟨*Get triangle vertices in* p1, p2, *and* p3⟩ ≡ 139, 140, 141, 145, 719
```
const Point &p1 = mesh->p[v[0]];
const Point &p2 = mesh->p[v[1]];
const Point &p3 = mesh->p[v[2]];
```

3.6.2 TRIANGLE INTERSECTION

An efficient algorithm for ray–triangle intersection can be derived using *barycentric coordinates*. Barycentric coordinates provide a way to parameterize a triangle in terms of two variables, b_1 and b_2:

$$\mathrm{p}(b_1, b_2) = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2.$$

The conditions on b_1 and b_2 are that $b_1 \geq 0$, $b_2 \geq 0$, and $b_1 + b_2 \leq 1$. The barycentric coordinates are also a natural way to interpolate across the surface of the triangle. Given values a_0, a_1, and a_2 defined at the vertices and given the barycentric coordinates for a point on the triangle, we can compute an interpolated value of a at that point as $(1 - b_1 - b_2)a_0 + b_1a_1 + b_2a_2$.

To derive an algorithm for intersecting a ray with a triangle, we insert the parametric ray equation into the triangle equation:

$$\mathrm{o} + t\mathbf{d} = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2. \tag{3.1}$$

Following the technique described by Möller and Trumbore (1997), we use the shorthand notation $\mathbf{e}_1 = \mathrm{p}_1 - \mathrm{p}_0$, $\mathbf{e}_2 = \mathrm{p}_2 - \mathrm{p}_0$, and $\mathbf{s} = \mathrm{o} - \mathrm{p}_0$. We can now rearrange the terms of Equation (3.1) to obtain the matrix equation

$$\begin{pmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2 \end{pmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \mathbf{s}. \tag{3.2}$$

Solving this linear system will give us both the barycentric coordinates of the intersection point as well as the parametric distance along the ray. Geometrically, we can interpret this system as a translation of the triangle to the origin and a transformation of the triangle to a unit right triangle in y and z.

We can easily solve Equation (3.2) using Cramer's rule. Note that we are introducing a bit of notation for brevity here; we write $|\ \mathbf{a} \quad \mathbf{b} \quad \mathbf{c}\ |$ to mean the determinant of the matrix having $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ as its columns. Cramer's rule gives

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{|\ -\mathbf{d} \quad \mathbf{e}_1 \quad \mathbf{e}_2\ |} \begin{bmatrix} |\ \mathbf{s} \quad \mathbf{e}_1 \quad \mathbf{e}_2| \\ |-\mathbf{d} \quad \mathbf{s} \quad \mathbf{e}_2| \\ |-\mathbf{d} \quad \mathbf{e}_1 \quad \mathbf{s}| \end{bmatrix}. \tag{3.3}$$

BBox 70
Point 63
Triangle 139
TriangleMesh::p 137

This can be rewritten using the identity $|\,\mathbf{a}\quad\mathbf{b}\quad\mathbf{c}\,| = -(\mathbf{a}\times\mathbf{c})\cdot\mathbf{b} = -(\mathbf{c}\times\mathbf{b})\cdot\mathbf{a}$. We then obtain

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{(\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{e}_1} \begin{bmatrix} (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{e}_2 \\ (\mathbf{d}\times\mathbf{e}_2)\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{e}_1)\cdot\mathbf{d} \end{bmatrix}. \tag{3.4}$$

If we use the substitution $\mathbf{s}_1 = \mathbf{d}\times\mathbf{e}_2$ and $\mathbf{s}_2 = \mathbf{s}\times\mathbf{e}_1$, the common subexpressions are made more explicit:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\mathbf{s}_1\cdot\mathbf{e}_1} \begin{bmatrix} \mathbf{s}_2\cdot\mathbf{e}_2 \\ \mathbf{s}_1\cdot\mathbf{s} \\ \mathbf{s}_2\cdot\mathbf{d} \end{bmatrix}. \tag{3.5}$$

In order to compute $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{s}$ we need 9 subtractions. To compute $\mathbf{s}_1$ and $\mathbf{s}_2$, we need 2 cross products, which is a total of 12 multiplications and 6 subtractions. Finally, to compute t, b_1, and b_2, we need 4 dot products (12 multiplications and 8 additions), 1 reciprocal, and 3 multiplications. Thus, the total cost of ray-triangle intersection is 1 divide, 27 multiplies, and 17 adds (counting adds and subtracts together). Note that some of these operations can be avoided if it is determined midcalculation that the ray does not intersect the triangle.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
bool Triangle::Intersect(const Ray &ray, float *tHit, float *rayEpsilon,
                         DifferentialGeometry *dg) const {
    ⟨Compute s1 141⟩
    ⟨Compute first barycentric coordinate 142⟩
    ⟨Compute second barycentric coordinate 142⟩
    ⟨Compute t to intersection point 142⟩
    ⟨Compute triangle partial derivatives 143⟩
    ⟨Interpolate (u, v) triangle parametric coordinates 143⟩
    ⟨Test intersection against alpha texture, if present 144⟩
    ⟨Fill in DifferentialGeometry from triangle hit 145⟩
    *tHit = t;
    *rayEpsilon = 1e-3f * *tHit;
    return true;
}
```

The implementation first computes the denominator from Equation (3.5). It finds the three mesh vertices that make up this particular `Triangle` and then computes the edge vectors and denominator. Note that if the denominator is zero, this triangle is degenerate and therefore cannot intersect a ray.

Cross() 62
DifferentialGeometry 102
Dot() 60
Ray 66
Triangle 139
Vector 57

⟨*Compute* $\mathbf{s}_1$⟩ ≡ 141
```
⟨Get triangle vertices in p1, p2, and p3 140⟩
Vector e1 = p2 - p1;
Vector e2 = p3 - p1;
Vector s1 = Cross(ray.d, e2);
float divisor = Dot(s1, e1);
```

```
if (divisor == 0.)
    return false;
float invDivisor = 1.f / divisor;
```

Now the routine can compute the barycentric coordinate b_1. Recall that barycentric coordinates that are less than zero or greater than one represent points outside the triangle, indicating no intersection.

⟨*Compute first barycentric coordinate*⟩ ≡ 141

```
Vector d = ray.o - p1;
float b1 = Dot(d, s1) * invDivisor;
if (b1 < 0. || b1 > 1.)
    return false;
```

The second barycentric coordinate, b_2, is computed in a similar way:

⟨*Compute second barycentric coordinate*⟩ ≡ 141

```
Vector s2 = Cross(d, e1);
float b2 = Dot(ray.d, s2) * invDivisor;
if (b2 < 0. || b1 + b2 > 1.)
    return false;
```

Now that we know the ray intersects the triangle, we compute the distance along the ray at which the intersection occurs. This gives a last opportunity to exit the procedure early, in the case where the t value falls outside the `Ray::mint` and `Ray::maxt` bounds.

⟨*Compute* `t` *to intersection point*⟩ ≡ 141

```
float t = Dot(e2, s2) * invDivisor;
if (t < ray.mint || t > ray.maxt)
    return false;
```

In order to generate consistent tangent vectors over triangle meshes, it is necessary to compute the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ using the parametric (u, v) values at the triangle vertices, if provided. Although the partial derivatives are the same at all points on the triangle, the implementation here recomputes them each time an intersection is found. Although this results in redundant computation, the storage savings for large triangle meshes can be significant.

The triangle is the set of points

$$\mathrm{p}_\mathrm{o} + u\frac{\partial \mathrm{p}}{\partial u} + v\frac{\partial \mathrm{p}}{\partial v},$$

for some p_o, where u and v range over the parametric coordinates of the triangle. We also know the three vertex positions p_i, $i = 1,\ 2,\ 3$, and the texture coordinates (u_i, v_i) at each vertex. From this it follows that the partial derivatives of p must satisfy

$$\mathrm{p}_i = \mathrm{p}_\mathrm{o} + u_i\frac{\partial \mathrm{p}}{\partial u} + v_i\frac{\partial \mathrm{p}}{\partial v}.$$

In other words, there is a unique affine mapping from the two-dimensional (u, v) space to points on the triangle (such a mapping exists even though the triangle is specified in 3D space because the triangle is planar). To compute expressions for $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$,

Cross() 62
Dot() 60
Ray::maxt 67
Ray::mint 67
Ray::o 67
Vector 57

we start by computing the differences $p_1 - p_3$ and $p_2 - p_3$, giving the matrix equation

$$\begin{pmatrix} u_1 - u_3 & v_1 - v_3 \\ u_2 - u_3 & v_2 - v_3 \end{pmatrix} \begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} p_1 - p_3 \\ p_2 - p_3 \end{pmatrix}.$$

Thus,

$$\begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} u_1 - u_3 & v_1 - v_3 \\ u_2 - u_3 & v_2 - v_3 \end{pmatrix}^{-1} \begin{pmatrix} p_1 - p_3 \\ p_2 - p_3 \end{pmatrix}.$$

Inverting a 2×2 matrix is straightforward. The inverse of the (u, v) differences matrix is

$$\frac{1}{(u_1 - u_3)(v_2 - v_3) - (v_1 - v_3)(u_2 - u_3)} \begin{pmatrix} v_2 - v_3 & -(v_1 - v_3) \\ -(u_2 - u_3) & u_1 - u_3 \end{pmatrix}.$$

⟨*Compute triangle partial derivatives*⟩ ≡ 141

```
Vector dpdu, dpdv;
float uvs[3][2];
GetUVs(uvs);
⟨Compute deltas for triangle partial derivatives 143⟩
float determinant = du1 * dv2 - dv1 * du2;
if (determinant == 0.f) {
    ⟨Handle zero determinant for triangle partial derivative matrix 143⟩
}
else {
    float invdet = 1.f / determinant;
    dpdu = ( dv2 * dp1 - dv1 * dp2) * invdet;
    dpdv = (-du2 * dp1 + du1 * dp2) * invdet;
}
```

⟨*Compute deltas for triangle partial derivatives*⟩ ≡ 143

```
float du1 = uvs[0][0] - uvs[2][0];
float du2 = uvs[1][0] - uvs[2][0];
float dv1 = uvs[0][1] - uvs[2][1];
float dv2 = uvs[1][1] - uvs[2][1];
Vector dp1 = p1 - p3, dp2 = p2 - p3;
```

Finally, it is necessary to handle the case when the matrix is singular and therefore cannot be inverted. Note that this only happens when the user-supplied per-vertex parameterization values are degenerate. In this case, the `Triangle` just chooses an arbitrary coordinate system about the triangle's surface normal, making sure that it is orthonormal:

⟨*Handle zero determinant for triangle partial derivative matrix*⟩ ≡ 143

```
CoordinateSystem(Normalize(Cross(e2, e1)), &dpdu, &dpdv);
```

CoordinateSystem() 63
Cross() 62
Triangle 139
Triangle::GetUVs() 144
Vector 57
Vector::Normalize() 63

To compute the (u, v) parametric coordinates at the hit point, the barycentric interpolation formula is applied to the (u, v) parametric coordinates at the vertices.

⟨*Interpolate* (u, v) *triangle parametric coordinates*⟩ ≡ 141

```
float b0 = 1 - b1 - b2;
float tu = b0*uvs[0][0] + b1*uvs[1][0] + b2*uvs[2][0];
float tv = b0*uvs[0][1] + b1*uvs[1][1] + b2*uvs[2][1];
```

The utility routine `GetUVs()` returns the (u, v) coordinates for the three vertices of the triangle, either from the `TriangleMesh`, if it has them, or returning default values if explicit (u, v) coordinates were not specified with the mesh.

⟨*Triangle Public Methods*⟩ +≡ 139

```
void GetUVs(float uv[3][2]) const {
    if (mesh->uvs) {
        uv[0][0] = mesh->uvs[2*v[0]];
        uv[0][1] = mesh->uvs[2*v[0]+1];
        uv[1][0] = mesh->uvs[2*v[1]];
        uv[1][1] = mesh->uvs[2*v[1]+1];
        uv[2][0] = mesh->uvs[2*v[2]];
        uv[2][1] = mesh->uvs[2*v[2]+1];
    }
    else {
        uv[0][0] = 0.; uv[0][1] = 0.;
        uv[1][0] = 1.; uv[1][1] = 0.;
        uv[2][0] = 1.; uv[2][1] = 1.;
    }
}
```

Before a successful intersection is reported, the intersection point is tested against an alpha mask texture, if one has been assigned to the shape. This texture can be thought of as a one-dimensional function over the triangle's surface, where at any point where its value is zero, the intersection is ignored, effectively treating that point on the triangle as not being present. (Chapter 10 defines the texture interface and implementations in more detail.) Alpha masks can be helpful for representing objects like leaves: a leaf can be modeled as a single triangle, with an alpha mask "cutting out" the edges so that a leaf shape remains. This functionality is much less useful for other primitives, so `pbrt` only supports it for triangles.

⟨*Test intersection against alpha texture, if present*⟩ ≡ 141

```
if (mesh->alphaTexture) {
    DifferentialGeometry dgLocal(ray(t), dpdu, dpdv,
                                 Normal(0,0,0), Normal(0,0,0),
                                 tu, tv, this);
    if (mesh->alphaTexture->Evaluate(dgLocal) == 0.f)
        return false;
}
```

Now we certainly have a hit and can update the values pointed to by the pointers passed to the intersection routine. We now have all the information we need to initialize the `DifferentialGeometry` structure for this intersection. In contrast to previous shapes, it's not necessary to transform the partial derivatives to world space, since the triangle's vertices were already transformed to world space. Like the disk, the partial derivatives of the triangle's normal are also both (0, 0, 0), since it is flat.

DifferentialGeometry 102
Normal 65
Texture::Evaluate() 520
TriangleMesh 135
TriangleMesh::
alphaTexture 137
TriangleMesh::uvs 137

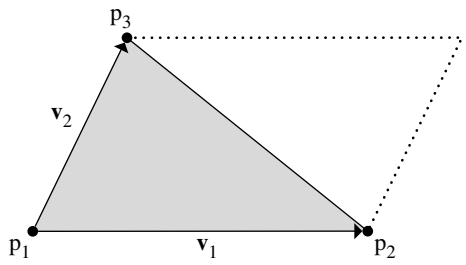


Figure 3.11: The area of a triangle with two edges given by vectors v_1 and v_2 is one-half of the area of the parallelogram. The parallelogram area is given by the length of the cross product of $\mathbf{v}_1$ and $\mathbf{v}_2$.

⟨*Fill in* `DifferentialGeometry` *from triangle hit*⟩ ≡ 141
```
*dg = DifferentialGeometry(ray(t), dpdu, dpdv,
                           Normal(0,0,0), Normal(0,0,0),
                           tu, tv, this);
```

3.6.3 SURFACE AREA

Recall from Section 2.2 that the area of a parallelogram is given by the length of the cross product of the two vectors along its sides. From this, it's easy to see that given the vectors for two edges of a triangle, its area is half of the area of the parallelogram given by those two vectors (Figure 3.11).

⟨*TriangleMesh Method Definitions*⟩ +≡
```
float Triangle::Area() const {
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    return 0.5f * Cross(p2-p1, p3-p1).Length();
}
```

3.6.4 SHADING GEOMETRY

If the `TriangleMesh` has per-vertex normals or tangent vectors, the `GetShadingGeometry()` method uses them to initialize the shading geometry appropriately:

⟨*TriangleMesh Method Definitions*⟩ +≡
```
void Triangle::GetShadingGeometry(const Transform &obj2world,
        const DifferentialGeometry &dg,
        DifferentialGeometry *dgShading) const {
    if (!mesh->n && !mesh->s) {
        *dgShading = dg;
        return;
    }
    ⟨Initialize Triangle shading geometry with n and s 146⟩
}
```

DifferentialGeometry 102
Normal 65
Transform 76
Triangle 139
Triangle::mesh 139
TriangleMesh 135
TriangleMesh::n 137
TriangleMesh::s 137

Because the DifferentialGeometry from a ray–triangle intersection stores parametric (u, v) coordinates but doesn't have the barycentric coordinates of the intersection point, the first task is to determine what the barycentric coordinates are.[5] These are then used to compute an interpolated normal vector and **s** vector, which are converted to two tangent vectors that give the same normal, since the DifferentialGeometry constructor takes two tangent vectors rather than the normal. Finally, the partial derivatives of the shading normal are found and the shading geometry structure can be initialized.

⟨*Initialize* Triangle *shading geometry with* n *and* s⟩ ≡ 145
```
    ⟨Compute barycentric coordinates for point 147⟩
    ⟨Use n and s to compute shading tangents for triangle, ss and ts 147⟩
    Normal dndu, dndv;
    ⟨Compute ∂n/∂u and ∂n/∂v for triangle shading geometry⟩
    *dgShading = DifferentialGeometry(dg.p, ss, ts,
        (*ObjectToWorld)(dndu), (*ObjectToWorld)(dndv),
        dg.u, dg.v, dg.shape);
```

Recall that the (u, v) parametric coordinates in the DifferentialGeometry for a triangle are computed with barycentric interpolation of parametric coordinates at the triangle vertices:

$$u = b_0u_0 + b_1u_1 + b_2u_2$$
$$v = b_0v_0 + b_1v_1 + b_2v_2,$$

where $b_0 = 1 - b_1 - b_2$. Here, the values of u, v, u_i, and v_i are all known, u and v from the DifferentialGeometry and u_i and v_i from the Triangle. We can substitute for the b_0 term and rewrite the above equations, giving a linear system in two unknowns b_1 and b_2:

$$\begin{pmatrix} u_1 - u_0 & u_2 - u_1 \\ v_1 - v_0 & v_2 - v_1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} u - u_0 \\ v - v_0 \end{pmatrix}.$$

This is a linear system of the basic form $\mathbf{AB} = \mathbf{C}$. We can solve for $\mathbf{B}$ by inverting $\mathbf{A}$, giving the two barycentric coordinates

$$\mathbf{B} = \mathbf{A}^{-1}\mathbf{C}.$$

The closed-form solution for linear systems of this form is implemented in the utility routine SolveLinearSystem2x2().

DifferentialGeometry 102
DifferentialGeometry::p 102
DifferentialGeometry::shape 102
DifferentialGeometry::u 102
DifferentialGeometry::v 102
Normal 65
Shape::ObjectToWorld 108
SolveLinearSystem2x2() 1020
Triangle 139

5 An alternative system design could recognize the importance of triangle meshes as a rendering primitive and provide space for the barycentric coordinates in the DifferentialGeometry structure, with other shapes not setting those fields. More generally, a handful of floats that could be used for shape-specific information like this could be provided in DifferentialGeometry. Other parts of the system wouldn't access these fields, not knowing their semantics, but they could be accessed by the shape in the GetShadingGeometry() call when DifferentialGeometry that it had initialized was passed back to it.

⟨*Compute barycentric coordinates for point*⟩ ≡ 146
```
float b[3];
⟨Initialize A and C matrices for barycentrics 147⟩
if (!SolveLinearSystem2x2(A, C, &b[1], &b[2])) {
    ⟨Handle degenerate parametric mapping 147⟩
}
else
    b[0] = 1.f - b[1] - b[2];
```

⟨*Initialize* A *and* C *matrices for barycentrics*⟩ ≡ 147
```
float uv[3][2];
GetUVs(uv);
float A[2][2] =
    { { uv[1][0] - uv[0][0], uv[2][0] - uv[0][0] },
      { uv[1][1] - uv[0][1], uv[2][1] - uv[0][1] } };
float C[2] = { dg.u - uv[0][0], dg.v - uv[0][1] };
```

If the determinant of **A** is zero, the solution is undefined and `SolveLinearSystem2x2()` returns `false`. This case happens if all three triangle vertices had the same texture coordinates, for example. In this case, the barycentric coordinates are all set arbitrarily to $\frac{1}{3}$:

⟨*Handle degenerate parametric mapping*⟩ ≡ 147
```
b[0] = b[1] = b[2] = 1.f/3.f;
```

Given the barycentrics, it's straightforward to compute the shading normal and shading **s** tangent vector by interpolating among the appropriate vertex normals and tangents. First, the vector `ts` for the differential geometry constructor is found using the cross product of `ss` and `ns`, giving an orthogonal vector to the two of them. Next, `ss` is overwritten with the cross product of `ns` and `ts`; this ensures that the cross product of `ss` and `ts` gives `ns`.

Thus, if per-vertex **n** and **s** values are provided and if the interpolated **n** and **s** values aren't perfectly orthogonal, **n** will be preserved and **s** will be modified so that the coordinate system is orthogonal.

⟨*Use* n *and* s *to compute shading tangents for triangle,* ss *and* ts⟩ ≡ 146
```
Normal ns;
Vector ss, ts;
if (mesh->n) ns = Normalize(obj2world(b[0] * mesh->n[v[0]] +
                                      b[1] * mesh->n[v[1]] +
                                      b[2] * mesh->n[v[2]]));
else   ns = dg.nn;
if (mesh->s) ss = Normalize(obj2world(b[0] * mesh->s[v[0]] +
                                      b[1] * mesh->s[v[1]] +
                                      b[2] * mesh->s[v[2]]));
else   ss = Normalize(dg.dpdu);
```

Cross() 62
DifferentialGeometry::u 102
DifferentialGeometry::v 102
Normal 65
SolveLinearSystem2x2() 1020
Vector 57
Vector::Normalize() 63

```
ts = Cross(ss, ns);
if (ts.LengthSquared() > 0.f) {
    ts = Normalize(ts);
    ss = Cross(ts, ns);
}
else
    CoordinateSystem((Vector)ns, &ss, &ts);
```

The code to compute the partial derivatives $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ is almost identical to the code to compute the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ on a triangle. Therefore, it has been elided from the text here.

⋆ 3.7 SUBDIVISION SURFACES

We will wrap up this chapter by defining a `Shape` that implements *subdivision surfaces*, a representation that is particularly well suited to describing complex smooth shapes. The subdivision surface for a particular mesh is defined by repeatedly subdividing the faces of the mesh into smaller faces and then finding the new vertex locations using weighted combinations of the old vertex positions.

For appropriately chosen subdivision rules, this process converges to give a smooth *limit surface* as the number of subdivision steps goes to infinity. In practice, just a few levels of subdivision typically suffice to give a good approximation of the limit surface. Figure 3.12 shows a simple example of a subdivision, where a tetrahedron has been subdivided zero, one, two, and six times. Figure 3.13 shows the effect of applying subdivision to the Killeroo model; on the top is the original control mesh, and below is the subdivision surface that the control mesh represents.

Although originally developed in the 1970s, subdivision surfaces have recently received a fair amount of attention in computer graphics thanks to some important advantages over

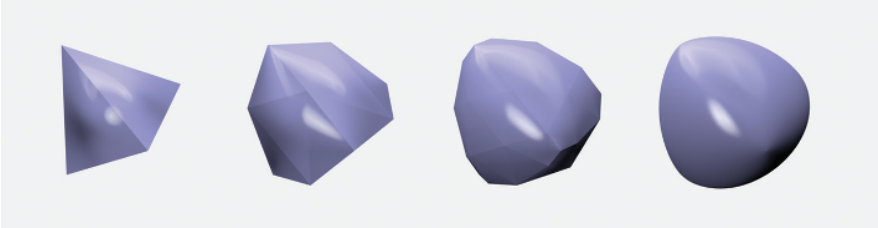


Figure 3.12: Subdivision of a Tetrahedron. From left to right, zero, one, two, and six subdivision steps have been used. (At zero levels, the vertices are just moved to lie on the limit surface.) As more subdivision is done, the mesh approaches the limit surface, the smooth surface described by the original mesh. Notice how the specular highlights become progressively more accurate and the silhouette edges appear smoother as more levels of subdivision are performed.

Shape 108

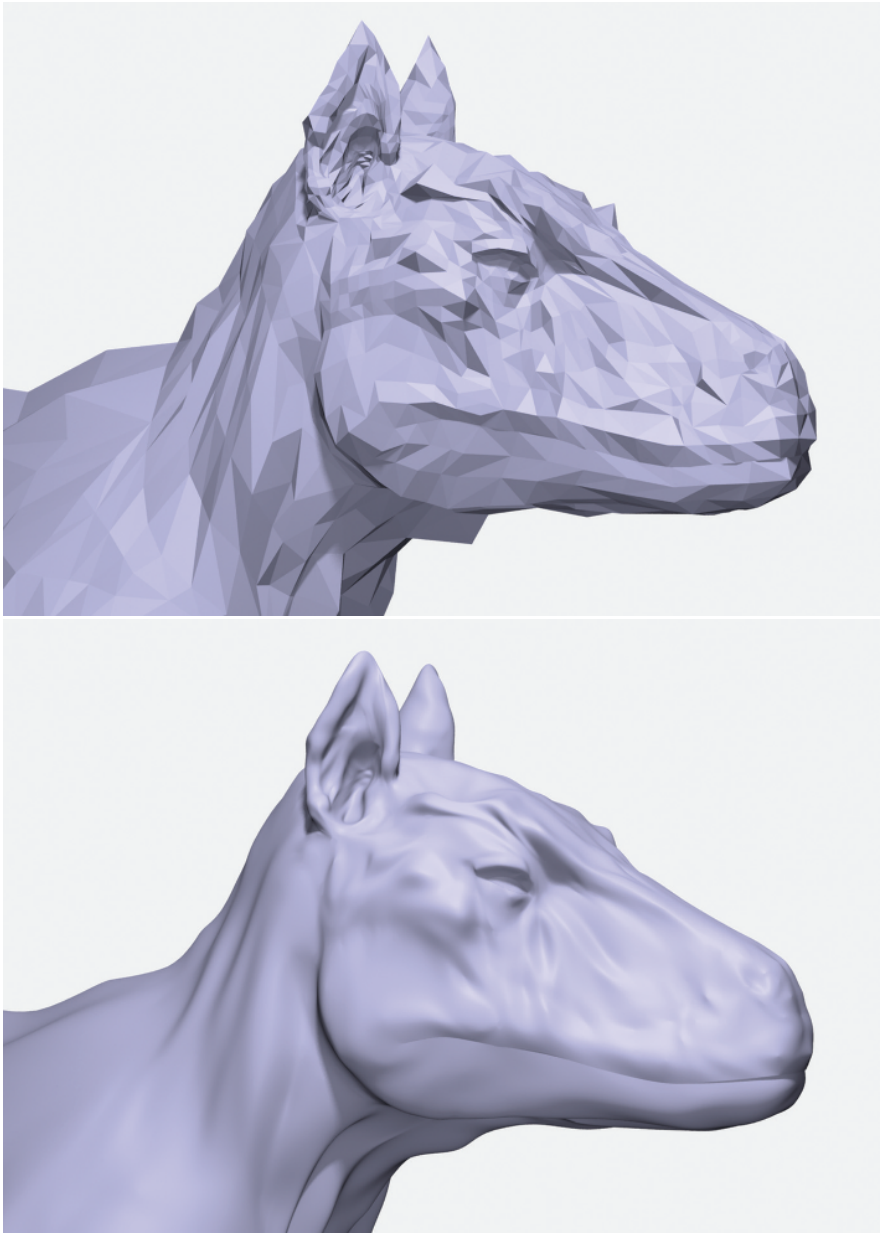


Figure 3.13: Subdivision Applied to the Killeroo Model. The control mesh (top) describes the subdivision surface shown below it. Subdivision is well suited to modeling shapes like this one, since it's easy to add detail locally by refining the control mesh, and there are no limitations on the topology of the final surface. *(Model courtesy of headus/Rezard.)*

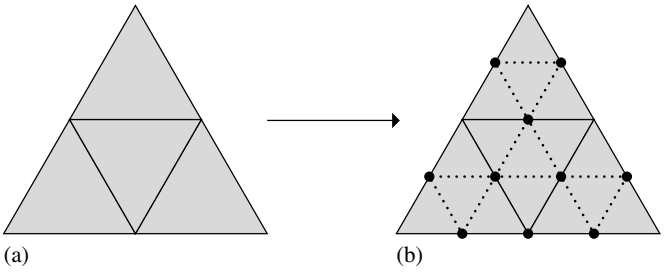


Figure 3.14: Basic Refinement Process for Loop Subdivision. (a) The control mesh before subdivision. (b) The new mesh after one subdivision step. Each triangular face of the mesh has been subdivided into four new faces by splitting each of the edges and connecting the new vertices with new edges.

polygonal and spline-based representations of surfaces. The advantages of subdivision include the following:

- Subdivision surfaces are smooth, as opposed to polygon meshes, which appear faceted when viewed close up, regardless of how finely they are modeled.
- A lot of existing infrastructure in modeling systems can be retargeted to subdivision. The classic toolbox of techniques for modeling polygon meshes can be applied to modeling subdivision control meshes.
- Subdivision surfaces are well suited to describing objects with complex topology, since they start with a control mesh of arbitrary (manifold) topology. Parametric surface models generally don't handle complex topology well.
- Subdivision methods are often generalizations of spline-based surface representations, so spline surfaces can often just be run through general subdivision surface renderers.
- It is easy to add detail to a localized region of a subdivision surface simply by adding faces to appropriate parts of the control mesh. This is much harder with spline representations.

Here, we will describe an implementation of *Loop subdivision surfaces*.[6] The Loop subdivision rules are based on triangular faces in the control mesh; faces with more than three vertices are triangulated at the start. At each subdivision step, all faces split into four child faces (Figure 3.14). New vertices are added along all of the edges of the original mesh, with positions computed using weighted averages of nearby vertices. Furthermore, the position of each original vertex is updated with a weighted average of its position and its new neighbors' positions. The implementation here uses weights based on improvements to Loop's method developed by Hoppe et al. (1994). We will not include discussion here about how these weights are derived. They must be chosen carefully to ensure that the limit surface actually has particular desirable smoothness properties, although subtle mathematics are necessary to prove that they indeed do this.

6 Named after the inventor of the subdivision rules used, Charles Loop.

3.7.1 MESH REPRESENTATION

⟨*LoopSubdiv Declarations*⟩ ≡
```
class LoopSubdiv : public Shape {
public:
    ⟨LoopSubdiv Public Methods⟩
private:
    ⟨LoopSubdiv Private Methods 163⟩
    ⟨LoopSubdiv Private Data 152⟩
};
```

We will start by describing the data structures used to represent the subdivision mesh. These data structures need to be carefully designed in order to support all of the operations necessary to cleanly implement the subdivision algorithm. The parameters to the `LoopSubdiv` constructor specify a triangle mesh in exactly the same format used in the `TriangleMesh` constructor (Section 3.6): each face is described by three integer vertex indices, giving offsets into the vertex array `P` for the face's three vertices. We will need to process this data to determine which faces are adjacent to each other, which faces are adjacent to which vertices, and so on.

⟨*LoopSubdiv Method Definitions*⟩ ≡
```
LoopSubdiv::LoopSubdiv(const Transform *o2w, const Transform *w2o,
                       bool ro, int nfaces, int nvertices,
                       const int *vertexIndices, const Point *P, int nl)
    : Shape(o2w, w2o, ro) {
    nLevels = nl;
    ⟨Allocate LoopSubdiv vertices and faces 151⟩
    ⟨Set face to vertex pointers 154⟩
    ⟨Set neighbor pointers in faces 156⟩
    ⟨Finish vertex initialization 158⟩
}
```

We will shortly define `SDVertex` and `SDFace` structures, which hold data for vertices and faces in the subdivision mesh. The constructor starts by allocating one instance of the `SDVertex` class for each vertex in the mesh and an `SDFace` for each face. For now, these are mostly uninitialized.

⟨*Allocate* `LoopSubdiv` *vertices and faces*⟩ ≡ 151
```
int i;
SDVertex *verts = new SDVertex[nvertices];
for (i = 0; i < nvertices; ++i) {
    verts[i] = SDVertex(P[i]);
    vertices.push_back(&verts[i]);
}
SDFace *fs = new SDFace[nfaces];
for (i = 0; i < nfaces; ++i)
    faces.push_back(&fs[i]);
```

LoopSubdiv 151
LoopSubdiv::faces 152
LoopSubdiv::vertices 152
Point 63
SDFace 153
SDVertex 152
Shape 108
Transform 76
TriangleMesh 135

The LoopSubdiv destructor, which we won't include here, deletes all of the faces and vertices allocated above.

⟨*LoopSubdiv Private Data*⟩ ≡ 151
```
int nLevels;
vector<SDVertex *> vertices;
vector<SDFace *> faces;
```

The Loop subdivision scheme, like most other subdivision schemes, assumes that the control mesh is *manifold*—no more than two faces share any given edge. Such a mesh may be closed or open: A *closed mesh* has no boundary, and all faces have adjacent faces across each of their edges. An *open mesh* has some faces that do not have all three neighbors. The LoopSubdiv implementation here supports both closed and open meshes.

In the interior of a triangle mesh, most vertices are adjacent to six faces and have six neighbor vertices directly connected to them with edges. On the boundaries of an open mesh, most vertices are adjacent to three faces and four vertices. The number of vertices directly adjacent to a vertex is called the vertex's *valence*. Interior vertices with valence other than six, or boundary vertices with valence other than four, are called *extraordinary vertices*; otherwise, they are called *regular*.[7] Loop subdivision surfaces are smooth everywhere except at their extraordinary vertices.

Each SDVertex stores its position P, a Boolean that indicates whether it is a regular or extraordinary vertex, and a Boolean that records if it lies on the boundary of the mesh. It also holds a pointer to an arbitrary face adjacent to it; this pointer gives a starting point for finding all of the adjacent faces. Finally, there is a pointer to store the corresponding SDVertex for the next level of subdivision, if any.

⟨*LoopSubdiv Local Structures*⟩ ≡
```
struct SDVertex {
    ⟨SDVertex Constructor 152⟩
    ⟨SDVertex Methods⟩
    Point P;
    SDFace *startFace;
    SDVertex *child;
    bool regular, boundary;
};
```

The constructor for SDVertex does the obvious initialization. Note that the value of SDVertex::startFace is initially set to NULL.

⟨*SDVertex Constructor*⟩ ≡ 152
```
SDVertex(Point pt = Point(0,0,0))
    : P(pt), startFace(NULL), child(NULL),
      regular(false), boundary(false) { }
```

LoopSubdiv 151
Point 63
SDFace 153
SDVertex 152
SDVertex::startFace 152

7 These terms are commonly used in the modeling literature, although "irregular" versus "regular" or "extraordinary" versus "ordinary" might be more intuitive.

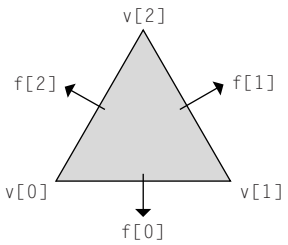


Figure 3.15: Each triangular face stores three pointers to `SDVertex` objects `v[i]` and three pointers to neighboring faces `f[i]`. Neighboring faces are indexed using the convention that the `i`th edge is the edge from `v[i]` to `v[(i+1)%3]`, and the neighbor across the `i`th edge is in `f[i]`.

The `SDFace` structure is where most of the topological information about the mesh is maintained. Because all faces are triangular, faces always store pointers to their three vertices and pointers to the adjacent faces across its three edges. The corresponding face neighbor pointers will be `NULL` if the face is on the boundary of an open mesh.

The face neighbor pointers are indexed such that if we label the edge from `v[i]` to `v[(i+1)%3]` as the `i`th edge, then the neighbor face across that edge is stored in `f[i]` (Figure 3.15). This labeling convention is important to keep in mind. Later when we are updating the topology of a newly subdivided mesh, we will make extensive use of it to navigate around the mesh. Similarly to the `SDVertex` class, the `SDFace` also stores pointers to child faces at the next level of subdivision.

⟨*LoopSubdiv Local Structures*⟩ +≡
```
struct SDFace {
    ⟨SDFace Constructor⟩
    ⟨SDFace Methods 159⟩
    SDVertex *v[3];
    SDFace *f[3];
    SDFace *children[4];
};
```

The `SDFace` constructor is straightforward—it simply sets these various pointers to `NULL`—so it is not shown here.

In order to simplify navigation of the `SDFace` data structure, we'll provide macros that make it easy to determine the vertex and face indices before or after a particular index. These macros add appropriate offsets and compute the result modulus three to handle cycling around. To compute the previous index, they add 2 instead of subtracting 1, which avoids possibly taking the modulus of a negative number, the result of which is implementation dependent in C++.

SDFace 153
SDVertex 152

⟨*LoopSubdiv Macros*⟩ ≡
```
#define NEXT(i) (((i)+1)%3)
#define PREV(i) (((i)+2)%3)
```

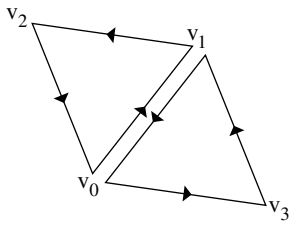


Figure 3.16: All of the faces in the input mesh must be specified so that each shared edge is given no more than once in each direction. Here, the edge from v_0 to v_1 is traversed from v_0 to v_1 by one face and from v_1 to v_0 by the other. Another way to think of this is in terms of face orientation: all faces' vertices should be given consistently in either clockwise or counterclockwise order, as seen from outside the mesh.

In addition to requiring a manifold mesh, the `LoopSubdiv` class expects that the control mesh specified by the user will be *consistently ordered*—each *directed edge* in the mesh can be present only once. An edge that is shared by two faces should be specified in a different direction by each face. Consider two vertices, v_0 and v_1, with an edge between them. We expect that one of the triangular faces that has this edge will specify its three vertices so that v_0 is before v_1, and that the other face will specify its vertices so that v_1 is before v_0 (Figure 3.16). A Möbius strip is one example of a surface that cannot be consistently ordered, but such surfaces come up rarely in rendering so in practice this restriction is not a problem. Poorly formed mesh data from other programs that don't create consistently ordered meshes can be troublesome, however.

Given this assumption about the input data, the constructor can now initialize this mesh's topological data structures. It first loops over all of the faces and sets their `v` pointers to point to their three vertices. It also sets each vertex's `SDVertex::startFace` pointer to point to one of the vertex's neighboring faces. It doesn't matter which of its adjacent faces is used, so the implementation just keeps resetting it each time it comes across another face that the vertex is incident to, thus ensuring that all vertices have some non-NULL face pointer by the time the loop is complete.

⟨*Set face to vertex pointers*⟩ ≡ 151

```
const int *vp = vertexIndices;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int j = 0; j < 3; ++j) {
        SDVertex *v = vertices[vp[j]];
        f->v[j] = v;
        v->startFace = f;
    }
    vp += 3;
}
```

LoopSubdiv 151
SDFace 153
SDFace::v 153
SDVertex 152
SDVertex::startFace 152

Now it is necessary to set each face's `f` pointer to point to its neighboring faces. This is a bit trickier, since face adjacency information isn't directly specified by the user. The constructor loops over the faces and creates an `SDEdge` object for each of their three edges. When it comes to another face that shares the same edge, it can update both faces' neighbor pointers.

```
⟨LoopSubdiv Local Structures⟩ +≡
    struct SDEdge {
        ⟨SDEdge Constructor 155⟩
        ⟨SDEdge Comparison Function 155⟩
        SDVertex *v[2];
        SDFace *f[2];
        int f0edgeNum;
    };
```

The `SDEdge` constructor takes pointers to the two vertices at each end of the edge. It orders them so that `v[0]` holds the one that is first in memory. This code may seem strange, but it is simply relying on the fact that pointers in C++ are effectively numbers that can be manipulated like integers,[8] and that the ordering of vertices on an edge is arbitrary. Sorting vertices on the address of the pointer guarantees that the edge (v_a, v_b) is correctly recognized as the same as the edge (v_b, v_a), regardless of what order the vertices are given in.

```
⟨SDEdge Constructor⟩ ≡ 155
    SDEdge(SDVertex *v0 = NULL, SDVertex *v1 = NULL) {
        v[0] = min(v0, v1);
        v[1] = max(v0, v1);
        f[0] = f[1] = NULL;
        f0edgeNum = -1;
    }
```

The class also defines an ordering operation for `SDEdge` objects so that they can be stored in other data structures that rely on ordering being well defined.

```
⟨SDEdge Comparison Function⟩ ≡ 155
    bool operator<(const SDEdge &e2) const {
        if (v[0] == e2.v[0]) return v[1] < e2.v[1];
        return v[0] < e2.v[0];
    }
```

Now the constructor can get to work, looping over the edges in all of the faces and updating the neighbor pointers as it goes. It uses a `set` to store the edges that have only one adjacent face so far. The `set` makes it possible to search for a particular edge in $O(\log n)$ time.

SDEdge 155
SDEdge::v 155
SDFace 153
SDVertex 152

8 Segmented architectures notwithstanding.

⟨*Set neighbor pointers in* faces⟩ ≡ 151
```
set<SDEdge> edges;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int edgeNum = 0; edgeNum < 3; ++edgeNum) {
        ⟨Update neighbor pointer for edgeNum 156⟩
    }
}
```

For each edge in each face, the loop body creates an edge object and sees if the same edge has been seen previously. If so, it initializes both faces' neighbor pointers across the edge. If not, it adds the edge to the set of edges. The indices of the two vertices at the ends of the edge, `v0` and `v1`, are equal to the edge index and the edge index plus one.

⟨*Update neighbor pointer for* edgeNum⟩ ≡ 156
```
int v0 = edgeNum, v1 = NEXT(edgeNum);
SDEdge e(f->v[v0], f->v[v1]);
if (edges.find(e) == edges.end()) {
    ⟨Handle new edge 156⟩
}
else {
    ⟨Handle previously seen edge 156⟩
}
```

Given an edge that hasn't been encountered before, the current face's pointer is stored in the edge object's `f[0]` member. Because the input mesh is assumed to be manifold, there can be at most one other face that shares this edge. When such a face is discovered, it can be used to initialize the neighboring face field. Storing the edge number of this edge in the current face allows the neighboring face to initialize its corresponding edge neighbor pointer.

⟨*Handle new edge*⟩ ≡ 156
```
e.f[0] = f;
e.f0edgeNum = edgeNum;
edges.insert(e);
```

When the second face on an edge is found, the neighbor pointers for each of the two faces are set. The edge is then removed from the edge set, since no edge can be shared by more than two faces.

⟨*Handle previously seen edge*⟩ ≡ 156
```
e = *edges.find(e);
e.f[0]->f[e.f0edgeNum] = f;
f->f[edgeNum] = e.f[0];
edges.erase(e);
```

Now that all faces have proper neighbor pointers, the `boundary` and `regular` flags in each of the vertices can be set. In order to determine if a vertex is a boundary vertex, we'll

NEXT() 153
SDEdge 155
SDEdge::f 155
SDEdge::f0edgeNum 155
SDFace 153
SDFace::f 153

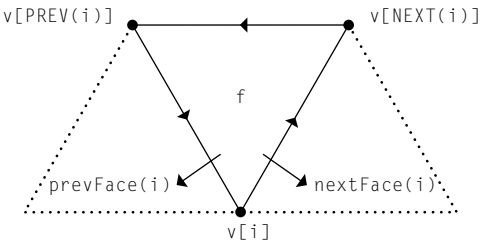


Figure 3.17: Given a vertex `v[i]` and a face that it is incident to, `f`, we define the *next face* as the face adjacent to `f` across the edge from `v[i]` to `v[NEXT(i)]`. The previous face is defined analogously.

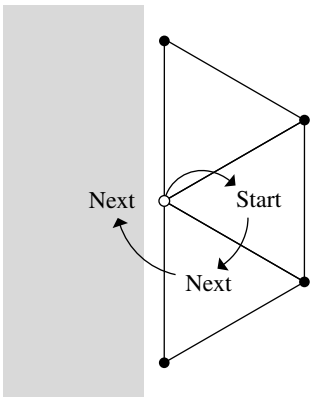


Figure 3.18: We can determine if a vertex is a boundary vertex by starting from the adjacent face `startFace` and following next face pointers around the vertex. If we come to a face that has no next neighbor face, then the vertex is on a boundary. If we return to `startFace`, it's an interior vertex.

define an ordering of faces around a vertex (Figure 3.17). For a vertex `v[i]` on a face `f`, we define the vertex's *next face* as the face across the edge from `v[i]` to `v[NEXT(i)]` and the *previous face* as the face across the edge from `v[PREV(i)]` to `v[i]`.

By successively going to the next face around `v`, we can iterate over the faces adjacent to it. If we eventually return to the face we started at, then we are at an interior vertex; if we come to an edge with a `NULL` neighbor pointer, then we're at a boundary vertex (Figure 3.18). Once the initialization routine has determined if this is a boundary vertex, it computes the valence of the vertex and sets the `regular` flag if the valence is 6 for an interior vertex or 4 for a boundary vertex; otherwise, it is an extraordinary vertex.

⟨*Finish vertex initialization*⟩ ≡ 151
```
for (i = 0; i < nvertices; ++i) {
    SDVertex *v = vertices[i];
    SDFace *f = v->startFace;
    do {
        f = f->nextFace(v);
    } while (f && f != v->startFace);
    v->boundary = (f == NULL);
    if (!v->boundary && v->valence() == 6)
        v->regular = true;
    else if (v->boundary && v->valence() == 4)
        v->regular = true;
    else
        v->regular = false;
}
```

Because the valence of a vertex is frequently needed, we provide the method `SDVertex::valence()`.

⟨*LoopSubdiv Inline Functions*⟩ ≡
```
inline int SDVertex::valence() {
    SDFace *f = startFace;
    if (!boundary) {
        ⟨Compute valence of interior vertex 158⟩
    }
    else {
        ⟨Compute valence of boundary vertex 159⟩
    }
}
```

To compute the valence of a nonboundary vertex, this method counts the number of the adjacent faces starting by following each face's neighbor pointers around the vertex until it reaches the starting face. The valence is equal to the number of faces visited.

⟨*Compute valence of interior vertex*⟩ ≡ 158
```
int nf = 1;
while ((f = f->nextFace(this)) != startFace)
    ++nf;
return nf;
```

For boundary vertices we can use the same approach, although in this case, the valence is one more than the number of adjacent faces. The loop over adjacent faces is slightly more complicated here: it follows pointers to the next face around the vertex until it reaches the boundary, counting the number of faces seen. It then starts again at `startFace` and follows previous face pointers until it encounters the boundary in the other direction.

SDFace 153
SDFace::nextFace() 159
SDVertex 152
SDVertex::boundary 152
SDVertex::regular 152
SDVertex::startFace 152
SDVertex::valence() 158

⟨*Compute valence of boundary vertex*⟩ ≡ 158
```
int nf = 1;
while ((f = f->nextFace(this)) != NULL)
    ++nf;
f = startFace;
while ((f = f->prevFace(this)) != NULL)
    ++nf;
return nf+1;
```

`SDFace::vnum()` is a utility function that finds the index of a given vertex pointer. It is a fatal error to pass a pointer to a vertex that isn't part of the current face—this case would represent a bug elsewhere in the subdivision code.

⟨*SDFace Methods*⟩ ≡ 153
```
int vnum(SDVertex *vert) const {
    for (int i = 0; i < 3; ++i)
        if (v[i] == vert) return i;
    Severe("Basic logic error in SDFace::vnum()");
    return -1;
}
```

Since the next face for a vertex `v[i]` on a face `f` is over the `i`th edge (recall the mapping of edge neighbor pointers from Figure 3.15), we can find the appropriate face neighbor pointer easily given the index `i` for the vertex, which the `vnum()` utility function provides. The previous face is across the edge from `PREV(i)` to `i`, so the method returns `f[PREV(i)]` for the previous face.

⟨*SDFace Methods*⟩ +≡ 153
```
SDFace *nextFace(SDVertex *vert) {
    return f[vnum(vert)];
}
```

⟨*SDFace Methods*⟩ +≡ 153
```
SDFace *prevFace(SDVertex *vert) {
    return f[PREV(vnum(vert))];
}
```

It is also useful to be able to get the next and previous vertices around a face starting at any vertex. The `SDFace::nextVert()` and `SDFace::prevVert()` methods do just that (Figure 3.19).

⟨*SDFace Methods*⟩ +≡ 153
```
SDVertex *nextVert(SDVertex *vert) {
    return v[NEXT(vnum(vert))];
}
```

⟨*SDFace Methods*⟩ +≡ 153
```
SDVertex *prevVert(SDVertex *vert) {
    return v[PREV(vnum(vert))];
}
```

NEXT() 153
PREV() 153
SDFace 153
SDFace::f 153
SDFace::nextFace() 159
SDFace::nextVert() 159
SDFace::prevFace() 159
SDFace::prevVert() 159
SDFace::v 153
SDFace::vnum() 159
SDVertex 152
Severe() 1005

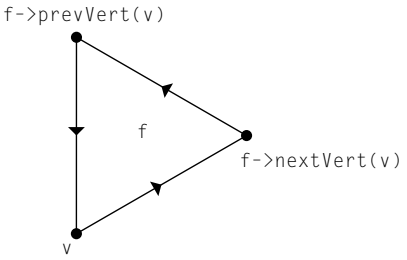


Figure 3.19: Given a vertex `v` on a face `f`, the method `f->prevVert(v)` returns the previous vertex around the face from `v`, and `f->nextVert(v)` returns the next vertex, where "next" and "previous" are defined by the original ordering of vertices when this face was defined.

3.7.2 BOUNDS

Loop subdivision surfaces have the *convex hull property*: the limit surface is guaranteed to be inside the convex hull of the original control mesh. Thus, for the bounding methods, we can just bound the original control vertices. The bounding methods are essentially equivalent to those in `TriangleMesh`, so we won't include them here.

3.7.3 SUBDIVISON

Now we can show how subdivision proceeds with the modified Loop rules. The `LoopSubdiv` shape doesn't support intersection directly, but applies subdivision a fixed number of times to generate a `TriangleMesh` for rendering. Exercise 3.10 at the end of the chapter discusses adaptive subdivision, where each original face is subdivided enough times so that the result looks smooth from a particular viewpoint rather than just using a fixed number of levels of subdivision, which may over-subdivide some areas while simultaneously under-subdividing others.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
bool LoopSubdiv::CanIntersect() const {
    return false;
}
```

The `Refine()` method handles all of the subdivision. It repeatedly applies the subdivision rules to the mesh, each time generating a new mesh to be used as the input to the next step. After each subdivision step, the `f` and `v` arrays in the `Refine()` method are updated to point to the faces and vertices from the level of subdivision just computed. When the `LoopSubdiv` is done subdividing, it creates a `TriangleMesh` representation of the surface that it returns to the caller. The `MemoryArena` class, defined in Section A.5.4 in Appendix A, provides a custom memory allocation method that quickly allocates memory, automatically freeing the memory when it goes out of scope.

LoopSubdiv 151
MemoryArena 1015
TriangleMesh 135

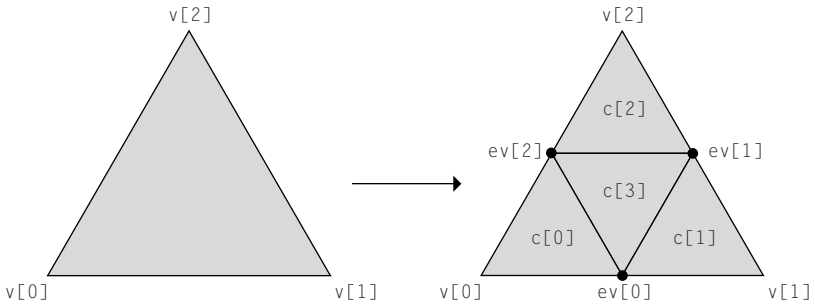


Figure 3.20: Basic Subdivision of a Single Triangular Face. Four child faces are created, ordered such that the `i`th child face is adjacent to the `i`th vertex of the original face and the fourth child face is in the center of the subdivided face. Three edge vertices need to be computed; they are numbered so that the `i`th edge vertex is along the `i`th edge of the original face.

⟨*LoopSubdiv Method Definitions*⟩ +≡

```
void LoopSubdiv::Refine(vector<Reference<Shape> > &refined) const {
    vector<SDFace *> f = faces;
    vector<SDVertex *> v = vertices;
    MemoryArena arena;
    for (int i = 0; i < nLevels; ++i) {
        ⟨Update f and v for next level of subdivision 161⟩
    }
    ⟨Push vertices to limit surface 171⟩
    ⟨Compute vertex tangents on limit surface 171⟩
    ⟨Create TriangleMesh from subdivision mesh⟩
}
```

The main loop of a subdivision step proceeds as follows: it creates `vector`s for all of the vertices and faces at the current level of subdivision and then proceeds to compute new vertex positions and update the topological representation for the refined mesh. Figure 3.20 shows the basic refinement rules for faces in the mesh. Each face is split into four child faces, such that the `i`th child face is next to the `i`th vertex of the input face and the final face is in the center. Three new vertices are then computed along the split edges of the original face.

LoopSubdiv 151
LoopSubdiv::faces 152
LoopSubdiv::nLevels 152
LoopSubdiv::vertices 152
MemoryArena 1015
Reference 1011
SDFace 153
SDVertex 152
Shape 108

⟨*Update* `f` *and* `v` *for next level of subdivision*⟩ ≡ 161

```
vector<SDFace *> newFaces;
vector<SDVertex *> newVertices;
⟨Allocate next level of children in mesh tree 162⟩
⟨Update vertex positions and create new edge vertices 162⟩
⟨Update new mesh topology 169⟩
⟨Prepare for next level of subdivision 170⟩
```

First, storage is allocated for the updated values of the vertices already present in the input mesh. The method also allocates storage for the child faces. It doesn't yet do any

initialization of the new vertices and faces other than setting the `regular` and `boundary` flags for the vertices since subdivision leaves boundary vertices on the boundary and interior vertices in the interior and it doesn't change the valence of vertices in the mesh.

⟨*Allocate next level of children in mesh tree*⟩ ≡ 161
```
for (uint32_t j = 0; j < v.size(); ++j) {
    v[j]->child = arena.Alloc<SDVertex>();
    v[j]->child->regular = v[j]->regular;
    v[j]->child->boundary = v[j]->boundary;
    newVertices.push_back(v[j]->child);
}
for (uint32_t j = 0; j < f.size(); ++j)
    for (int k = 0; k < 4; ++k) {
        f[j]->children[k] = arena.Alloc<SDFace>();
        newFaces.push_back(f[j]->children[k]);
    }
```

Computing New Vertex Positions

Before worrying about initializing the topology of the subdivided mesh, the refinement method computes positions for all of the vertices in the mesh. First, it considers the problem of computing updated positions for all of the vertices that were already present in the mesh; these vertices are called *even vertices*. It then computes the new vertices on the split edges. These are called *odd vertices.*

⟨*Update vertex positions and create new edge vertices*⟩ ≡ 161
```
⟨Update vertex positions for even vertices 162⟩
⟨Compute new odd edge vertices 166⟩
```

Different techniques are used to compute the updated positions for each of the different types of even vertices—regular and extraordinary, boundary and interior. This gives four cases to handle.

⟨*Update vertex positions for even vertices*⟩ ≡ 162
```
for (uint32_t j = 0; j < v.size(); ++j) {
    if (!v[j]->boundary) {
        ⟨Apply one-ring rule for even vertex 163⟩
    }
    else {
        ⟨Apply boundary rule for even vertex 165⟩
    }
}
```

For both types of interior vertices, we take the set of vertices adjacent to each vertex (called the *one-ring* around it, reflecting the fact that it's a ring of neighbors) and weight each of the neighbor vertices by a weight β (Figure 3.21). The vertex we are updating, in the center, is weighted by $1 - n\beta$, where n is the valence of the vertex. Thus, the new position v' for a vertex v is

$$\mathrm{v}' = (1 - n\beta)\mathrm{v} + \sum_{i=1}^{N} \beta \mathrm{v}_i.$$

MemoryArena::Alloc() 1016
SDFace 153
SDFace::children 153
SDVertex 152
SDVertex::boundary 152
SDVertex::regular 152

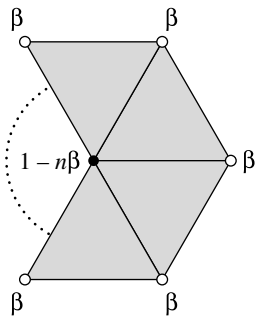


Figure 3.21: The new position v′ for a vertex v is computed by weighting the adjacent vertices v_i by a weight β and weighting v by $(1 - n\beta)$, where n is the valence of v. The adjacent vertices v_i are collectively referred to as the *one-ring* around v.

This formulation ensures that the sum of weights is one, which guarantees the convex hull property that was used earlier for bounding the surface. The position of the vertex being updated is only affected by vertices that are nearby; this is known as *local support*. Loop subdivision is particularly efficient because its subdivision rules all have this property.

The specific weight β used for this step is a key component of the subdivision method and must be chosen carefully in order to ensure smoothness of the limit surface, among other desirable properties.[9] The `LoopSubdiv::beta()` method that follows computes a β value based on the vertex's valence that ensures smoothness. For regular interior vertices, `LoopSubdiv::beta()` returns $\frac{1}{16}$. Since this is a common case, the implementation uses $\frac{1}{16}$ directly instead of calling `LoopSubdiv::beta()` every time.

⟨*Apply one-ring rule for even vertex*⟩ ≡ 162
```
if (v[j]->regular)
    v[j]->child->P = weightOneRing(v[j], 1.f/16.f);
else
    v[j]->child->P = weightOneRing(v[j], beta(v[j]->valence()));
```

⟨*LoopSubdiv Private Methods*⟩ ≡ 151
```
static float beta(int valence) {
    if (valence == 3) return 3.f/16.f;
    else return 3.f / (8.f * valence);
}
```

LoopSubdiv::beta() 171
LoopSubdiv::weightOneRing() 164
SDVertex::child 152
SDVertex::P 152
SDVertex::regular 152
SDVertex::valence() 158

9 Again, see the papers cited at the start of this section and in the "Further Reading" section for information about how values like β are derived.

The `LoopSubdiv::weightOneRing()` function loops over the one-ring of adjacent vertices and applies the given weight to compute a new vertex position. It uses the `SDVertex::oneRing()` function, defined in the following, which returns the positions of the vertices around the vertex `vert`.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
Point LoopSubdiv::weightOneRing(SDVertex *vert, float beta) {
    ⟨Put vert one-ring in Pring 164⟩
    Point P = (1 - valence * beta) * vert->P;
    for (int i = 0; i < valence; ++i)
        P += beta * Pring[i];
    return P;
}
```

Because a variable number of vertices are in the one-rings, we use the `ALLOCA()` macro to efficiently allocate space to store their positions.

⟨*Put* `vert` *one-ring in* `Pring`⟩ ≡ 164, 166
```
int valence = vert->valence();
Point *Pring = ALLOCA(Point, valence);
vert->oneRing(Pring);
```

The `oneRing()` method assumes that the pointer passed in points to an area of memory large enough to hold the one-ring around the vertex.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
void SDVertex::oneRing(Point *P) {
    if (!boundary) {
        ⟨Get one-ring vertices for interior vertex 164⟩
    }
    else {
        ⟨Get one-ring vertices for boundary vertex 165⟩
    }
}
```

It's relatively easy to get the one-ring around an interior vertex by looping over the faces adjacent to the vertex and for each face retaining the vertex after the center vertex. (Brief sketching with pencil and paper should convince you that this process returns all of the vertices in the one-ring.)

⟨*Get one-ring vertices for interior vertex*⟩ ≡ 164
```
SDFace *face = startFace;
do {
    *P++ = face->nextVert(this)->P;
    face = face->nextFace(this);
} while (face != startFace);
```

The one-ring around a boundary vertex is a bit more tricky. The implementation here carefully stores the one-ring in the given `Point` array so that the first and last entries in the array are the two adjacent vertices along the boundary. This ordering is important

ALLOCA() 1009
LoopSubdiv 151
LoopSubdiv::weightOneRing() 164
Point 63
SDFace 153
SDFace::nextFace() 159
SDFace::nextVert() 159
SDVertex 152
SDVertex::boundary 152
SDVertex::oneRing() 164
SDVertex::P 152
SDVertex::startFace 152
SDVertex::valence() 158

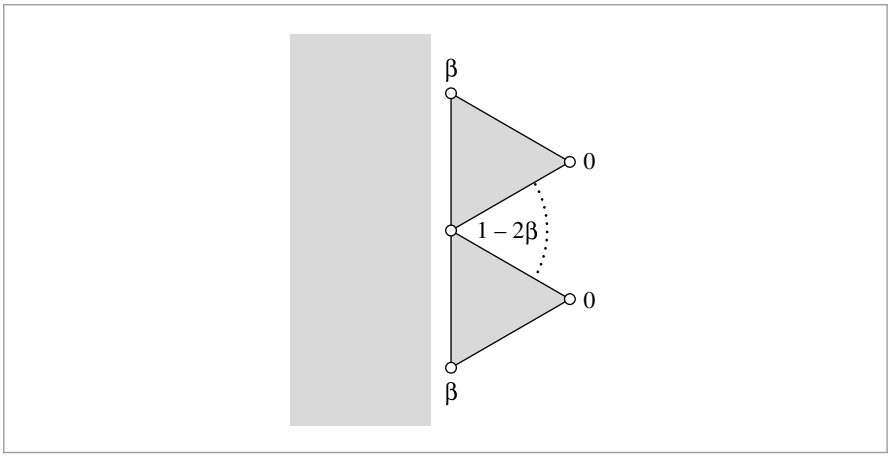


Figure 3.22: Subdivision on a Boundary Edge. The new position for the vertex in the center is computed by weighting it and its two neighbor vertices by the weights shown.

because the adjacent boundary vertices will often be weighted differently than the adjacent vertices that are in the interior of the mesh. Doing so requires that we first loop around neighbor faces until we reach a face on the boundary and then loop around the other way, storing vertices one by one.

⟨*Get one-ring vertices for boundary vertex*⟩ ≡ 164

```
SDFace *face = startFace, *f2;
while ((f2 = face->nextFace(this)) != NULL)
    face = f2;
*P++ = face->nextVert(this)->P;
do {
    *P++ = face->prevVert(this)->P;
    face = face->prevFace(this);
} while (face != NULL);
```

For vertices on the boundary, the new vertex's position is based only on the two neighboring boundary vertices (Figure 3.22). Not depending on interior vertices ensures that two abutting surfaces that share the same vertices on the boundary will have abutting limit surfaces. The `weightBoundary()` utility function applies the given weighting on the two neighbor vertices v_1 and v_2 to compute the new position v' as

$$v' = (1 - 2\beta)v + \beta v_1 + \beta v_2.$$

The same weight of $\frac{1}{8}$ is used for both regular and extraordinary vertices.

⟨*Apply boundary rule for even vertex*⟩ ≡ 162

```
v[j]->child->P = weightBoundary(v[j], 1.f/8.f);
```

The `weightBoundary()` utility function applies the given weights at a boundary vertex. Because the `SDVertex::oneRing()` function orders the boundary vertex's one-ring such

LoopSubdiv::
weightBoundary() 166
SDFace 153
SDFace::nextFace() 159
SDFace::nextVert() 159
SDFace::prevFace() 159
SDFace::prevVert() 159
SDVertex::child 152
SDVertex::oneRing() 164
SDVertex::P 152
SDVertex::startFace 152

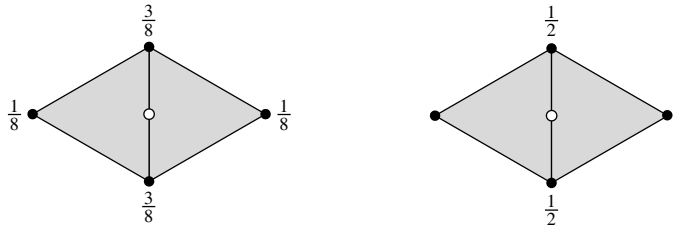


Figure 3.23: Subdivision Rule for Edge Split. The position of the new odd vertex, marked with an open circle, is found by weighting the two vertices at the ends of the edge and the two vertices opposite it on the adjacent triangles. (a) The weights for an interior vertex; (b) the weights for a boundary vertex.

that the first and last entries are the boundary neighbors, the implementation here is particularly straightforward.

⟨*LoopSubdiv Method Definitions*⟩ +≡
```
Point LoopSubdiv::weightBoundary(SDVertex *vert, float beta) {
    ⟨Put vert one-ring in Pring 164⟩
    Point P = (1-2*beta) * vert->P;
    P += beta * Pring[0];
    P += beta * Pring[valence-1];
    return P;
}
```

Now the refinement method computes the positions of the odd vertices—the new vertices along the split edges of the mesh. It loops over each edge of each face in the mesh, computing the new vertex that splits the edge (Figure 3.23). For interior edges, the new vertex is found by weighting the two vertices at the ends of the edge and the two vertices across from the edge on the adjacent faces. It loops through all three edges of each face, and each time it comes to an edge that hasn't been seen before it computes and stores the new odd vertex for the edge in the `edgeVerts` associative array.

⟨*Compute new odd edge vertices*⟩ ≡ 162
```
map<SDEdge, SDVertex *> edgeVerts;
for (uint32_t j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Compute odd vertex on kth edge 167⟩
    }
}
```

LoopSubdiv 151
Point 63
SDEdge 155
SDFace 153
SDVertex 152
SDVertex::P 152

As was done when setting the face neighbor pointers in the original mesh, an `SDEdge` object is created for the edge and checked to see if it is in the set of edges that have already

been visited. If it isn't, the new vertex on this edge is computed and added to the `map`, which is an associative array structure that performs efficient lookups.

⟨*Compute odd vertex on* `k`*th edge*⟩ ≡ 166
```
SDEdge edge(face->v[k], face->v[NEXT(k)]);
SDVertex *vert = edgeVerts[edge];
if (!vert) {
    ⟨Create and initialize new odd vertex 167⟩
    ⟨Apply edge rules to compute new vertex position 167⟩
    edgeVerts[edge] = vert;
}
```

In Loop subdivision, the new vertices added by subdivision are always regular. (This means that the proportion of extraordinary vertices with respect to regular vertices will decrease with each level of subdivision.) Therefore, the `regular` member of the new vertex can immediately be set to `true`. The `boundary` member can also be easily initialized, by checking to see if there is a neighbor face across the edge that is being split. Finally, the new vertex's `startFace` pointer can also be set here. For all odd vertices on the edges of a face, the center child (child face number three) is guaranteed to be adjacent to the new vertex.

⟨*Create and initialize new odd vertex*⟩ ≡ 167
```
vert = arena.Alloc<SDVertex>();
newVertices.push_back(vert);
vert->regular = true;
vert->boundary = (face->f[k] == NULL);
vert->startFace = face->children[3];
```

For odd boundary vertices, the new vertex is just the average of the two adjacent vertices. For odd interior vertices, the two vertices at the ends of the edge are given weight $\frac{3}{8}$, and the two vertices opposite the edge are given weight $\frac{1}{8}$ (Figure 3.23). These last two vertices can be found using the `SDFace::otherVert()` utility function, which returns the vertex opposite a given edge of a face.

⟨*Apply edge rules to compute new vertex position*⟩ ≡ 167
```
if (vert->boundary) {
    vert->P =  0.5f * edge.v[0]->P;
    vert->P += 0.5f * edge.v[1]->P;
}
else {
    vert->P =  3.f/8.f * edge.v[0]->P;
    vert->P += 3.f/8.f * edge.v[1]->P;
    vert->P += 1.f/8.f * face->otherVert(edge.v[0], edge.v[1])->P;
    vert->P += 1.f/8.f *
        face->f[k]->otherVert(edge.v[0], edge.v[1])->P;
}
```

MemoryArena::Alloc() 1016
NEXT() 153
SDEdge 155
SDEdge::v 155
SDFace::children 153
SDFace::f 153
SDFace::otherVert() 168
SDFace::v 153
SDVertex 152
SDVertex::boundary 152
SDVertex::P 152
SDVertex::regular 152
SDVertex::startFace 152

The `SDFace::otherVert()` method is self-explanatory:

⟨*SDFace Methods*⟩ +≡ 153

```
SDVertex *otherVert(SDVertex *v0, SDVertex *v1) {
    for (int i = 0; i < 3; ++i)
        if (v[i] != v0 && v[i] != v1)
            return v[i];
    Severe("Basic logic error in SDVertex::otherVert()");
    return NULL;
}
```

Updating Mesh Topology

In order to keep the details of the topology update as straightforward as possible, the numbering scheme for the subdivided faces and their vertices has been chosen carefully (Figure 3.24). Review that figure carefully; the conventions shown there are key to the next few pages.

There are four main tasks required to update the topological pointers of the refined mesh:

1. The odd vertices' `SDVertex::startFace` pointers need to store a pointer to one of their adjacent faces.
2. Similarly, the even vertices' `SDVertex::startFace` pointers must be set.
3. The new faces' neighbor `f[i]` pointers need to be set to point to the neighboring faces.
4. The new faces' `v[i]` pointers need to point to the appropriate vertices.

The `startFace` pointers of the odd vertices were already initialized when they were first created. We'll handle the other three tasks in order here.

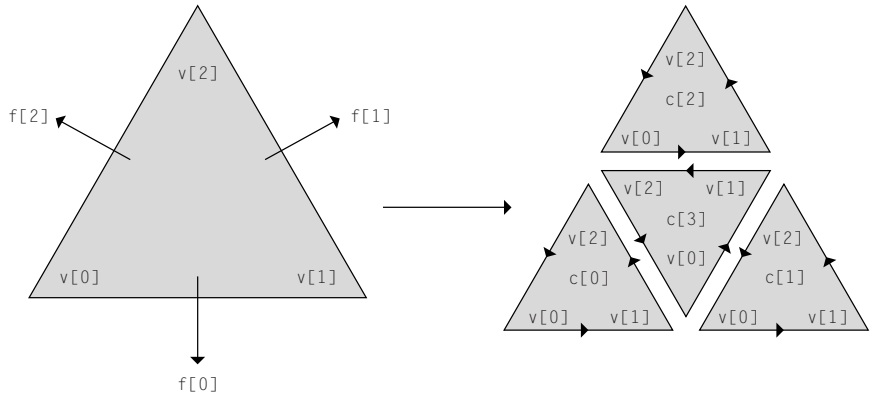


Figure 3.24: Each face is split into four child faces, such that the `i`th child is adjacent to the `i`th vertex of the original face, and such that the `i`th child face's `i`th vertex is the child of the `i`th vertex of the original face. The vertices of the center child are oriented such that the `i`th vertex is the odd vertex along the `i`th edge of the parent face.

SDFace::otherVert() 168
SDFace::v 153
SDVertex 152
SDVertex::startFace 152
Severe() 1005

⟨*Update new mesh topology*⟩ ≡ 161
⟨*Update even vertex face pointers* 169⟩
⟨*Update face neighbor pointers* 169⟩
⟨*Update face vertex pointers* 170⟩

If a vertex is the `i`th vertex of its `startFace`, then it is guaranteed that it will be adjacent to the `i`th child face of `startFace`. Therefore, it is just necessary to loop through all the parent vertices in the mesh, and for each one find its vertex index in its `startFace`. This index can then be used to find the child face adjacent to the new even vertex.

⟨*Update even vertex face pointers*⟩ ≡ 169
```
for (uint32_t j = 0; j < v.size(); ++j) {
    SDVertex *vert = v[j];
    int vertNum = vert->startFace->vnum(vert);
    vert->child->startFace =
        vert->startFace->children[vertNum];
}
```

Next, the face neighbor pointers for the newly created faces are updated. We break this into two steps: one to update neighbors among children of the same parent, and one to do neighbors across children of different parents. This involves some tricky pointer manipulation.

⟨*Update face neighbor pointers*⟩ ≡ 169
```
for (uint32_t j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update children f pointers for siblings 169⟩
        ⟨Update children f pointers for neighbor children 170⟩
    }
}
```

For the first step, recall that the interior child face is always stored in `children[3]`. Furthermore, the `k` + 1st child face (for `k` = 0, 1, 2) is across the `k`th edge of the interior face, and the interior face is across the `k` + 1st edge of the `k`th face.

⟨*Update children* `f` *pointers for siblings*⟩ ≡ 169
```
face->children[3]->f[k] = face->children[NEXT(k)];
face->children[k]->f[NEXT(k)] = face->children[3];
```

We'll now update the children's face neighbor pointers that point to children of other parents. Only the first three children need to be addressed here; the interior child's neighbor pointers have already been fully initialized. Inspection of Figure 3.24 reveals that the `k`th and `PREV(k)`th edges of the `i`th child need to be set. To set the `k`th edge of the `k`th child, we first find the `k`th edge of the parent face, then the neighbor parent `f2` across that edge. If `f2` exists (meaning we aren't on a boundary), the neighbor parent index for the vertex `v[k]` is found. That index is equal to the index of the neighbor child we are searching for. This process is then repeated to find the child across the `PREV(k)`th edge.

NEXT() 153
SDFace 153
SDFace::children 153
SDFace::f 153
SDFace::vnum() 159
SDVertex 152
SDVertex::startFace 152

⟨*Update children* f *pointers for neighbor children*⟩ ≡ 169
```
SDFace *f2 = face->f[k];
face->children[k]->f[k] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
f2 = face->f[PREV(k)];
face->children[k]->f[PREV(k)] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
```

Finally, we handle the fourth step in the topological updates: setting the children faces' vertex pointers.

⟨*Update face vertex pointers*⟩ ≡ 169
```
for (uint32_t j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update child vertex pointer to new even vertex 170⟩
        ⟨Update child vertex pointer to new odd vertex 170⟩
    }
}
```

For the kth child face (for $k = 0, 1, 2$), the kth vertex corresponds to the even vertex that is adjacent to the child face. For the noninterior child faces, there is one even vertex and two odd vertices; for the interior child face, there are three odd vertices. This vertex can be found by following the child pointer of the parent vertex, available from the parent face.

⟨*Update child vertex pointer to new even vertex*⟩ ≡ 170
```
face->children[k]->v[k] = face->v[k]->child;
```

To update the rest of the vertex pointers, the `edgeVerts` associative array is reused to find the odd vertex for each split edge of the parent face. Three child faces have that vertex as an incident vertex. The vertex indices for the three faces are easily found, again based on the numbering scheme established in Figure 3.24.

⟨*Update child vertex pointer to new odd vertex*⟩ ≡ 170
```
SDVertex *vert = edgeVerts[SDEdge(face->v[k], face->v[NEXT(k)])];
face->children[k]->v[NEXT(k)] = vert;
face->children[NEXT(k)]->v[k] = vert;
face->children[3]->v[k] = vert;
```

After the geometric and topological work has been done for a subdivision step, the newly created vertices and faces are moved into the `v` and `f` arrays:

⟨*Prepare for next level of subdivision*⟩ ≡ 161
```
f = newFaces;
v = newVertices;
```

To the Limit Surface and Output

One of the remarkable properties of subdivision surfaces is that there are special subdivision rules that give the positions that the vertices of the mesh would have if we continued subdividing forever. We apply these rules here to initialize an array of limit surface positions, `Plimit`. Note that it's important to temporarily store the limit surface positions

NEXT() 153
PREV() 153
SDEdge 155
SDFace 153
SDFace::children 153
SDFace::f 153
SDFace::v 153
SDFace::vnum() 159
SDVertex 152
SDVertex::child 152

somewhere other than in the vertices while the computation is taking place. Because the limit surface position of each vertex depends on the original positions of its surrounding vertices, the original positions of all vertices must remain unchanged until the computation is complete.

The limit rule for a boundary vertex weights the two neighbor vertices by $\frac{1}{5}$ and the center vertex by $\frac{3}{5}$. The rule for interior vertices is based on a function `gamma()`, which computes appropriate vertex weights based on the valence of the vertex.

⟨*Push vertices to limit surface*⟩ ≡ 161
```
Point *Plimit = new Point[v.size()];
for (uint32_t i = 0; i < v.size(); ++i) {
    if (v[i]->boundary)
        Plimit[i] =  weightBoundary(v[i], 1.f/5.f);
    else
        Plimit[i] =  weightOneRing(v[i], gamma(v[i]->valence()));
}
for (uint32_t i = 0; i < v.size(); ++i)
    v[i]->P = Plimit[i];
```

⟨*LoopSubdiv Private Methods*⟩ +≡ 151
```
static float gamma(int valence) {
    return 1.f / (valence + 3.f / (8.f * beta(valence)));
}
```

In order to generate a smooth-looking triangle mesh with per-vertex surface normals, a pair of nonparallel tangent vectors to the limit surface is computed at each vertex. As with the limit rule for positions, this is an analytic computation that gives the precise tangents on the actual limit surface.

⟨*Compute vertex tangents on limit surface*⟩ ≡ 161
```
vector<Normal> Ns;
Ns.reserve(v.size());
vector<Point> Pring(16, Point());
for (uint32_t i = 0; i < v.size(); ++i) {
    SDVertex *vert = v[i];
    Vector S(0,0,0), T(0,0,0);
    int valence = vert->valence();
    if (valence > (int)Pring.size())
        Pring.resize(valence);
    vert->oneRing(&Pring[0]);
    if (!vert->boundary) {
        ⟨Compute tangents of interior face 172⟩
    } else {
        ⟨Compute tangents of boundary face 173⟩
    }
    Ns.push_back(Normal(Cross(S, T)));
}
```

Cross() 62
LoopSubdiv::gamma() 171
LoopSubdiv::
 weightBoundary() 166
LoopSubdiv::
 weightOneRing() 164
Normal 65
Point 63
SDVertex 152
SDVertex::boundary 152
SDVertex::oneRing() 164
SDVertex::valence() 158
Vector 57

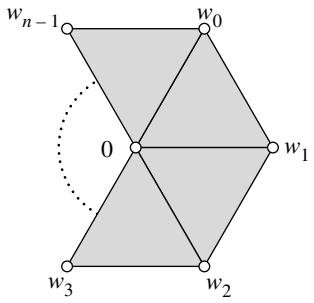


Figure 3.25: To compute tangents for interior vertices, the one-ring vertices are weighted with weights w_i. The center vertex, where the tangent is being computed, always has a weight of 0.

Figure 3.25 shows the setting for computing tangents in the mesh interior. The center vertex is given a weight of zero and the neighbors are given weights w_i. To compute the first tangent vector **s**, the weights are

$$w_i = \cos\left(\frac{2\pi i}{n}\right),$$

where n is the valence of the vertex. The second tangent **t** is computed with weights

$$w_i = \sin\left(\frac{2\pi i}{n}\right).$$

⟨*Compute tangents of interior face*⟩ ≡ 171

```
for (int k = 0; k < valence; ++k) {
    S += cosf(2.f*M_PI*k/valence) * Vector(Pring[k]);
    T += sinf(2.f*M_PI*k/valence) * Vector(Pring[k]);
}
```

Tangents on boundary vertices are a bit trickier. Figure 3.26 shows the ordering of vertices in the one-ring expected in the following discussion.

The first tangent, known as the *across tangent*, is given by the vector between the two neighboring boundary vertices:

$$\mathbf{s} = \mathrm{v}_{n-1} - \mathrm{v}_0.$$

The second tangent, known as the *transverse tangent*, is computed based on the vertex's valence. The center vertex is given a weight w_c and the one-ring vertices are given weights specified by a vector $(w_0, w_1, \ldots, w_{n-1})$. The transverse tangent rules we will use are

Valence	w_c	w_i
2	−2	(1, 1)
3	−1	(0, 1, 0)
4 (regular)	−2	(−1, 2, 2, −1)

M_PI 1002
Vector 57

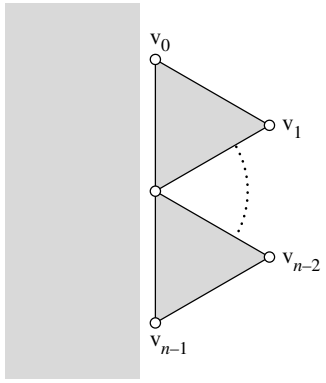


Figure 3.26: Tangents at boundary vertices are also computed as weighted averages of the adjacent vertices. However, some of the boundary tangent rules incorporate the value of the center vertex.

For valences of 5 and higher, $w_c = 0$ and

$$w_0 = w_{n-1} = \sin\theta$$
$$w_i = (2\cos\theta - 2)\ \sin(\theta i),$$

where

$$\theta = \frac{\pi}{n-1}.$$

Although we will not prove it here, these weights sum to zero for all values of i. This guarantees that the weighted sum is in fact a tangent vector.

⟨*Compute tangents of boundary face*⟩ ≡ 171

```
S = Pring[valence-1] - Pring[0];
if (valence == 2)
    T = Vector(Pring[0] + Pring[1] - 2 * vert->P);
else if (valence == 3)
    T = Pring[1] - vert->P;
else if (valence == 4) // regular
    T = Vector(-1*Pring[0] + 2*Pring[1] + 2*Pring[2] +
               -1*Pring[3] + -2*vert->P);
else {
    float theta = M_PI / float(valence-1);
    T = Vector(sinf(theta) * (Pring[0] + Pring[valence-1]));
    for (int k = 1; k < valence-1; ++k) {
        float wt = (2*cosf(theta) - 2) * sinf((k) * theta);
        T += Vector(wt * Pring[k]);
    }
    T = -T;
}
```

M_PI 1002
Vector 57

Finally, the fragment ⟨*Create* `TriangleMesh` *from subdivision mesh*⟩ creates the triangle mesh object and adds it to the `refined` vector passed to the `LoopSubdiv::Refine()` method. We won't include it here, since it's just a straightforward transformation of the subdivided mesh into an indexed triangle mesh.

FURTHER READING

An Introduction to Ray Tracing has an extensive survey of algorithms for ray-shape intersection (Glassner 1989a). Goldstein and Nagel (1971) discussed ray-quadric intersections, and Heckbert (1984) discussed the mathematics of quadrics for graphics applications in detail, with many citations to literature in mathematics and other fields. Hanrahan (1983) described a system that automates the process of deriving a ray intersection routine for surfaces defined by implicit polynomials; his system emits C source code to perform the intersection test and normal computation for a surface described by a given equation. Mitchell (1990) showed that interval arithmetic could be applied to develop algorithms for robustly computing intersections with implicit surfaces that cannot be described by polynomials and are thus more difficult to accurately compute intersections for; more recent work in this area was done by Knoll et al. (2009). See Moore's book (1966) for an introduction to interval arithmetic.

Other notable early papers related to ray–shape intersection include Kajiya's work on computing intersections with surfaces of revolution and procedurally generated fractal terrains (Kajiya 1983). Fournier et al.'s paper on rendering procedural stochastic models (Fournier, Fussel, and Carpenter 1982) and Hart et al.'s paper on finding intersections with fractals (Hart, Sandin, and Kauffman 1989) illustrate the broad range of shape representations that can be used with ray-tracing algorithms.

Kajiya (1982) developed the first algorithm for computing intersections with parametric patches. Recent work on more efficient techniques for direct ray intersection with patches includes papers by Stürzlinger (1998), Martin et al. (2000), and Roth et al. (2001). Benthin et al. (2004) presented more recent results and include additional references to previous work.

The ray–triangle intersection test in Section 3.6 was developed by Möller and Trumbore (1997). A ray–quadrilateral intersection routine was developed by Lagae and Dutré (2005). Shevtsov et al. (2007a) described a highly optimized ray–triangle intersection routine for modern CPU architectures and included a number of references to other recent approaches. An interesting approach for developing a fast ray–triangle intersection routine was introduced by Kensler and Shirley (2006): they implemented a program that performed a search across the space of mathematically equivalent ray–triangle tests, automatically generating software implementations of variations and then benchmarking them. In the end, they found an more efficient ray–triangle routine than had been in use previously.

The layout of triangle meshes in memory can have a measurable impact on performance in many situations. In general, if triangles that are close together in 3D space are close together in memory, cache hit rates will be higher, and overall system performance will

benefit. See Yoon et al. (2005) and Yoon and Lindstrom (2006) for algorithms for creating cache-friendly mesh layouts in memory.

Subdivision surfaces were invented by Doo and Sabin (1978) and Catmull and Clark (1978). The Loop subdivision method was originally developed by Charles Loop (1987), although the implementation in `pbrt` uses the improved rules for subdivision and tangents along boundary edges developed by Hoppe et al. (1994). There has been extensive work in subdivision surfaces recently. The SIGGRAPH course notes give a good summary of the state of the art and also have extensive references (Zorin et al. 2000). See also Warren's book on the topic (Warren 2002). Müller et al. (2003) described an approach that refines a subdivision surface on demand for the rays to be tested for intersection with it. (See also Benthin et al. 2007, for a related approach.)

An exciting development in subdivision surfaces is the ability to evaluate them at arbitrary points on the surface (Stam 1998). Subdivision surface implementations like the one in this chapter are often relatively inefficient, spending as much time dereferencing pointers as they do applying subdivision rules. Stam's approach also reduces the impact of this problem. Bolz and Schröder (2002) suggest a much more efficient implementation approach that precomputes a number of quantities that make it possible to compute the final mesh much more efficiently. More recently, Patney et al. (2009) have demonstrated a very efficient approach for tessellating subdivision surfaces on data-parallel throughput processors.

Phong and Crow (1975) first introduced the idea of interpolating per-vertex shading normals to give the appearance of smooth surfaces from polygonal meshes.

The notion of shapes that repeatedly refine themselves into collections of other shapes until ready for rendering was first introduced in the Reyes renderer (Cook, Carpenter, and Catmull 1987).

An excellent introduction to differential geometry is Gray (1993); Section 14.3 of that book presents the Weingarten equations. Turkowski (1990a) has expressions for first and second partial derivatives of a handful of parametric primitives.

EXERCISES

2 3.1 One nice property of mesh-based shapes like triangle meshes and subdivision surfaces is that the shape's vertices can be transformed into world space, so that it isn't necessary to transform rays into object space before performing ray intersection tests. Interestingly enough, it is possible to do the same thing for ray–quadric intersections.

The implicit forms of the quadrics in this chapter were all of the form

$$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + G = 0,$$

where some of the constants $A \ldots G$ were zero. More generally, we can define quadric surfaces by the equation

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gz + 2Hy + 2Iz + J = 0,$$

where most of the parameters $A \ldots J$ don't directly correspond to the earlier $A \ldots G$. In this form, the quadric can be represented by a 4×4 symmetric matrix $\mathbf{Q}$:

$$[\, x \quad y \quad z \quad 1 \,] \begin{pmatrix} A & D & F & G \\ D & B & E & H \\ F & E & C & I \\ G & H & I & J \end{pmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathrm{p}^T \mathbf{Q} \mathrm{p} = 0.$$

Given this representation, first show that the matrix $\mathbf{Q}'$ representing a quadric transformed by the matrix $\mathbf{M}$ is

$$\mathbf{Q}' = (\mathbf{M}^T)^{-1} \mathbf{Q} \mathbf{M}^{-1}.$$

To do so, show that for any point p where $\mathrm{p}^T \mathbf{Q} \mathrm{p} = 0$, if we apply a transformation $\mathbf{M}$ to p and compute $\mathrm{p}' = \mathbf{M}\mathrm{p}$, we'd like to find $\mathbf{Q}'$ so that $(\mathrm{p}')^T \mathbf{Q}' \mathrm{p}' = 0$.

Next, substitute the ray equation into the earlier more general quadric equation to compute coefficients for the quadratic equation $at^2 + bt + c = 0$ in terms of entries of the matrix $\mathbf{Q}$ to pass to the `Quadratic()` function.

Now implement this approach in `pbrt` and use it instead of the original quadric intersection routines. Note that you will still need to transform the resulting world space hit points into object space to test against $\theta_{\max}$, if it is not 2π, and so on. How does performance compare to the original scheme?

❶ 3.2 Improve the object space bounding box routines for the quadrics to properly account for $\phi_{\max} < 2\pi$ and compute tighter bounding boxes when possible. How much does this improve performance when rendering scenes with partial quadric shapes?

❷ 3.3 There is room to optimize the implementations of the various quadric primitives in `pbrt` in a number of ways. For example, for complete spheres some of the tests in the intersection routine related to partial spheres are unnecessary. Furthermore, some of the quadrics have calls to trigonometric functions that could be turned into simpler expressions using insight about the geometry of the particular primitives. Investigate ways to speed up these methods. How much does doing so improve the overall run time of `pbrt`?

❶ 3.4 Currently `pbrt` recomputes the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ for triangles every time they are needed, even though they are constant for each triangle. Precompute these vectors and analyze the speed/storage trade-off, especially for large triangle meshes. How does the depth complexity of the scene and the size of triangles in the image affect this trade-off?

❷ 3.5 Implement a general polygon primitive that supports an arbitrary number of vertices and convex or concave polygons as a new `Shape` in `pbrt`. You can assume that a valid polygon has been provided and that all of the vertices of the polygon lie on the same plane, although you might want to issue a warning when this is not the case.

Quadratic() 118
Shape 108

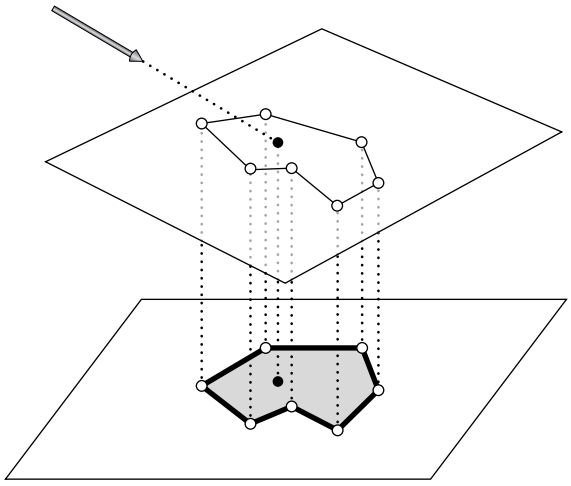


Figure 3.27: A ray–polygon intersection test can be performed by finding the point where the ray intersects the polygon's plane, projecting the hit point and polygon vertices onto an axis-aligned plane, and doing a 2D point-in-polygon test there.

An efficient technique for computing ray-polygon intersections is to find the plane equation for the polygon from its normal and a point on the plane. Then compute the intersection of the ray with that plane and project the intersection point and the polygon vertices to 2D. Then apply a 2D point-in-polygon test to determine if the point is inside the polygon. An easy way to do this is to effectively do a 2D ray-tracing computation and intersect the ray with each of the edge segments and count how many it goes through. If it goes through an odd number of them, the point is inside the polygon and there is an intersection. See Figure 3.27 for an illustration of this idea.

You may find it helpful to read the article by Haines (1994) that surveys a number of approaches for efficient point-in-polygon tests. Some of the techniques described there may be helpful for optimizing this test. Furthermore, Section 13.3.3 of Schneider and Eberly (2003) discusses strategies for getting all the corner cases right, for example, when the 2D ray is aligned precisely with an edge or passes through a vertex of the polygon.

2 3.6 Constructive solid geometry (CSG) is a classic solid modeling technique, where complex shapes are built up by considering the union, intersection, and differences of more primitive shapes. For example, a sphere could be used to create pits in a cylinder if a shape was modeled as the difference of a cylinder and set of spheres that partially overlapped it. See Hoffmann (1989) for further information about CSG.

Add support for CSG to pbrt and render images that demonstrate interesting shapes that can be rendered using CSG. You may want to read Roth (1982), which first described how ray tracing could be used to render models described by CSG, as well as Amanatides and Mitchell (1990), which discusses precision-related issues for CSG ray tracing.

● 3.7 Procedurally–described parametric surfaces: Write a Shape that takes a general mathematical expression of the form $f(u, v) \rightarrow (x, y, z)$ that describes a parametric surface as a function of (u, v). Evaluate the given function at a grid of (u, v) positions and create a TriangleMesh that approximates the given surface when the Shape::Refine() method is called.

■ 3.8 Almost all methods for subdivision surfaces are based on either refining a mesh of triangles or a mesh of quadrilaterals. If a rendering system only supports one type of mesh, meshes of the other type are typically tessellated to make faces of the expected type in a preprocessing step. However, doing this can introduce artifacts in the final subdivision surface. Read Stam and Loop's paper on a hybrid subdivision scheme that supports meshes with both quadrilateral and triangular faces (Stam and Loop 2003) and implement a Shape based on their method. Demonstrate cases where the subdivision surface that your implementation creates does not have artifacts that are present in the output from LoopSubdiv.

● 3.9 The smoothness of subdivision surfaces isn't always desirable. Sometimes it is useful to be able to flag some edges of a subdivision control mesh as "creases" and apply different subdivision rules there to preserve a sharp edge. Extend the LoopSubdiv implementation so that some edges can be denoted as creases, and use the boundary subdivision rules to compute the positions of vertices along those edges. Render images showing the difference this makes.

■ 3.10 Implement adaptive subdivision for the LoopSubdiv Shape. A weakness of the basic implementation in Section 3.7 is that each face is always refined a fixed number of times: this may mean that some faces are underrefined, leading to visible faceting in the triangle mesh, and some faces are overrefined, leading to excessive memory use and rendering time. With adaptive subdivision, individual faces are no longer subdivided once a particular error threshold has been reached.

An easy error threshold to implement computes the face normals of each face and its directly adjacent faces. If they are sufficiently close to each other (e.g., as tested via dot products), then the limit surface for that face will be reasonably flat and further refinement will likely make little difference to the final surface. Alternatively, you might want to approximate the area that a subdivided face covers on the image plane and continue subdividing until this area becomes sufficiently small. This approximation could be done using ray differentials; see Section 10.1.1 for an explanation of how to relate the ray differential to the screen space footprint.

LoopSubdiv 151
Shape 108
Shape::Refine() 110
TriangleMesh 135

The trickiest part of this exercise is that some faces that don't need subdivision due to the flatness test will still need to be subdivided in order to provide vertices so that neighboring faces that do need to subdivide can get their vertex one-rings. In particular, adjacent faces can differ by no more than one level of subdivision. You may find it useful to read recent papers by Patney et al. (2009) and Fisher et al. (2009) for discussion of how to avoid cracks in adaptively subdivided meshes.

3.11 Ray-tracing point-sampled geometry: Extending methods for rendering complex models represented as a collection of point samples (Levoy and Whitted 1985; Pfister et al. 2000; Rusinkiewicz and Levoy 2000), Schaufler and Jensen (2000) have described a method for intersecting rays with collections of oriented point samples in space. They probabilistically determined that an intersection has occurred when a ray approaches a sufficient local density of point samples and computes a surface normal with a weighted average of the nearby samples. Read their paper and extend `pbrt` to support a point-sampled geometry shape. Do any of `pbrt`'s basic interfaces need to be extended or generalized to support a shape like this?

3.12 Ray-tracing ribbons: Hair is often modeled as a collection of *generalized cylinders*, which are defined as the shape that results from sweeping a disk along a given curve. Because there are often a large number of individual hairs, an efficient method for intersecting rays with generalized cylinders is needed for ray-tracing hair. A number of methods have been developed to compute ray intersections with generalized cylinders (Bronsvoort and Klok 1985; de Voogt, van der Helm, and Bronsvoort 2000); investigate these algorithms and extend `pbrt` to support a fast hair primitive with one of them. Alternatively, investigate the generalization of Schaufler and Jensen's approach for probabilistic point intersection (Schaufler and Jensen 2000) to probabilistic line intersection and apply this to fast ray tracing of hair.

3.13 Deformation motion blur: The `TransformedPrimitive` in Section 4.1.2 of Chapter 4 supports animated shapes via transformations of primitives that vary over time. However, this type of animation isn't general enough to represent a triangle mesh where each vertex has a position given at the start time and another one at the end time. (For example, this type of animation description can be used to describe a running character model where different parts of the body are moving in different ways, such that the overall shape deforms.) Implement a more general `TriangleMesh` shape that supports specifying vertex positions at the start and end of frame and interpolates between them based on the ray time passed to the intersection methods. Be sure to update the bounding routines appropriately.

Meshes with very large amounts of motion may exhibit poor performance due to triangles sweeping out very large bounding boxes and thus many ray-triangle intersections being performed that don't hit the triangle. Can you come up with approaches that could be used to reduce the impact of this problem?

TransformedPrimitive 190

3.14 Implicit functions: Just as implicit definitions of the quadric shapes are a useful starting point for deriving ray intersection algorithms, more complex implicit functions can also be used to define interesting shapes. In particular, difficult-to-model organic shapes, water drops, and so on can be well-represented by implicit surfaces. Blinn (1982a) introduced the idea of directly rendering implicit surfaces, and Wyvill and Wyvill (1989) gave a basis function for implicit surfaces with a number of advantages compared to Blinn's.

Implement a method for finding ray intersections with general implicit surfaces and add it to `pbrt`. You may wish to read papers by Kalra and Barr (1989) and Hart (1996) for methods for ray-tracing them. Mitchell's algorithm for robust ray intersections with implicit surfaces gives another effective method for finding these intersections (Mitchell 1990), and more recently Knoll et al. (2009) described refinements to this idea. You may find an approach along these lines easier to implement than the others. See Moore's book on interval arithmetic as needed for reference (Moore 1966).

3.15 L-systems: A very successful technique for procedurally modeling plants was introduced to graphics by Alvy Ray Smith (1984), who applied *Lindenmayer systems* (L-systems) to model branching plant structures. Prusinkiewicz and collaborators have generalized this approach to encompass a much wider variety of types of plants and effects that determine their appearance (Prusinkiewicz 1986; Prusinkiewicz, James, and Mech 1994; Deussen et al. 1998; Prusinkiewicz et al. 2001). L-systems describe the branching structure of these types of shapes via a grammar. The grammar can be evaluated to form expressions that describe a topological representation of a plant, which can then be translated into a geometric representation. Add an L-system primitive to `pbrt` that takes a grammar as input and evaluates it on demand at rendering time to create the plant described by it.

3.16 The expressions used to initialize `rayEpsilon` values in the intersection routines in this chapter were derived in an *ad hoc* manner and aren't necessarily always sufficiently conservative. For either the ray–triangle intersection routine or some of the ray–quadric intersection routines, apply techniques from numerical analysis to derive conservative bounds on the floating-point error in the computed intersection parametertic t values. In general, these expressions should depend on the particular floating-point computations performed in the intersection routines and will be related to the magnitudes of the ray origin, direction, and the parameters that describe the shape (vertex position, radius, etc.).

3.17 As an alternative to Exercise 3.16, Kalra and Barr (1989) described an approach for robust ray–object intersections that is based on recursively subdividing the bounding box of the object, discarding boxes that don't encompass the object's surface, and discarding boxes that the ray misses. Dammertz and Keller (2006) proposed an application of this approach where once a box has been found where the ray intersects it and it is no longer possible to subdivide the box without it vanishing due to no remaining numerical precision, then an intersection

is reported. (They also described implementations for spline patches and triangles.) Implement this approach in `pbrt` and evaluate its speed and quality. You will probably need to make substantial modifications to the system, as the `Shape::Intersect()` method probably should return a pair of `Point`s, one on the side of the surface the ray came from and the other on the other side of the surface, rather than just a parametric t value. (These two points then should be be used for tracing spawned reflected and transmitted rays, respectively.)

Point 63
Shape::Intersect() 111

CHAPTER FOUR



04 PRIMITIVES AND INTERSECTION ACCELERATION

The classes described in the last chapter focus exclusively on representing geometric properties of 3D objects. Although the `Shape` class provides a convenient abstraction for geometric operations such as intersection and bounding, it doesn't contain enough information to fully describe an object in a scene. For example, it is necessary to bind material properties to each shape in order to specify its appearance. To accomplish these goals, this chapter introduces the `Primitive` class and provides a number of implementations.

Shapes to be rendered directly are represented by the `GeometricPrimitive` class. This class combines a `Shape` with a description of its appearance properties. So that the geometric and shading portions of `pbrt` can be cleanly separated, these appearance properties are encapsulated in the `Material` class, which is described in Chapter 9.

The `TransformedPrimitive` class handles two more general uses of `Shapes` in the scene: shapes with animated transformation matrices and object instancing, which can greatly reduce the memory requirements for scenes that contain many instances of the same geometry at different locations (such as the one in Figure 4.1). Implementing each of these features essentially requires injecting an additional transformation matrix between the `Shape`'s notion of world space and the actual scene world space. Therefore, both are handled by a single class.

This chapter also introduces the `Aggregate` base class, which represents a container that can hold many `Primitives`. `pbrt` uses this class to implement *acceleration structures*—data structures that help reduce the otherwise $O(n)$ complexity of testing a ray for intersection with all n objects in a scene. Most rays will intersect only a few primitives and miss the others by a large distance. If an intersection acceleration technique can reject



Figure 4.1: This ecosystem scene makes heavy use of instancing as a mechanism for compressing the scene's description. There are only 1.1 million unique triangles in the scene, although, thanks to object reuse through instancing, the total geometric complexity is 19.5 million triangles.

whole groups of primitives at once, there will be a substantial performance improvement compared to simply testing each ray against each primitive in turn. One benefit from reusing the `Primitive` interface for these acceleration structures is that `pbrt` can support hybrid approaches where an accelerator of one type holds accelerators of other types.

This chapter describes the implementation of three accelerators, one (`GridAccel`) based on overlaying a uniform grid over the scene, one (`BVHAccel`) based on building a hierarchy of bounding boxes around objects in the scene, and the last (`KdTreeAccel`) based on adaptive recursive spatial subdivision.

4.1 PRIMITIVE INTERFACE AND GEOMETRIC PRIMITIVES

`BVHAccel` 209
`GridAccel` 196
`KdTreeAccel` 228
`Primitive` 185
`ReferenceCounted` 1010

The abstract `Primitive` base class is the bridge between the geometry processing and shading subsystems of `pbrt`. It inherits from the `ReferenceCounted` base class, which automatically tracks how many references there are to an object, freeing its storage when the last reference goes out of scope. Other classes that store `Primitives` shouldn't store pointers to them, but instead hold a `Reference<Primitive>`, which ensures that reference

counts are computed correctly. The `Reference<Primitive>` class otherwise behaves as if it was a pointer to a `Primitive`.

⟨*Primitive Declarations*⟩ ≡
```
class Primitive : public ReferenceCounted {
public:
    ⟨Primitive Interface 185⟩
    ⟨Primitive Public Data 185⟩
protected:
    ⟨Primitive Protected Data 185⟩
};
```

Like `Shapes`, `Primitives` also each have a unique 32-bit identifier. Because a single `Shape` may be represented in the scene multiple times due to object instancing, both the shape and primitive ids are necessary to uniquely identify an instance of a shape in the scene.

⟨*Primitive Interface*⟩ ≡ 185
```
Primitive() : primitiveId(nextprimitiveId++) { }
```

⟨*Primitive Public Data*⟩ ≡ 185
```
const uint32_t primitiveId;
```

⟨*Primitive Protected Data*⟩ ≡ 185
```
static uint32_t nextprimitiveId;
```

Similarly to shape ids, the first primitive id value handed out is one, so that zero can be reserved to indicate "no primitive."

⟨*Primitive Method Definitions*⟩ ≡
```
uint32_t Primitive::nextprimitiveId = 1;
```

Because the `Primitive` class connects geometry and shading, its interface contains methods related to both. There are five geometric routines in the `Primitive` interface, all of which are similar to a corresponding `Shape` method. The first, `Primitive::WorldBound()`, returns a box that encloses the primitive's geometry in world space. There are many uses for such a bound; one of the most important is to place the `Primitive` in the acceleration data structures.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual BBox WorldBound() const = 0;
```

`BBox` 70
`DifferentialGeometry` 102
`Intersection` 186
`Primitive` 185
`Primitive::CanIntersect()` 186
`Primitive::WorldBound()` 185
`ReferenceCounted` 1010
`Shape` 108

Similarly to the `Shape` class, all primitives must be able to either determine if a given ray intersects their geometry or else refine themselves into one or more new primitives. Like the `Shape` interface, `Primitive` has a `Primitive::CanIntersect()` method so that `pbrt` can determine whether the underlying geometry is intersectable or not.

One difference from the `Shape` interface is that the `Primitive` intersection methods return `Intersection` structures rather than `DifferentialGeometry`. These `Intersection` structures hold more information about the intersection than just the local geometric information, such as information about the material properties at the hit point.

Another difference is that `Shape::Intersect()` returns the parametric distance along the ray to the intersection in a `float *` output variable, while `Primitive::Intersect()` is responsible for updating `Ray::maxt` with this value if an intersection is found.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual bool CanIntersect() const;
virtual bool Intersect(const Ray &r, Intersection *in) const = 0;
virtual bool IntersectP(const Ray &r) const = 0;
virtual void Refine(vector<Reference<Primitive> > &refined) const;
```

The `Intersection` structure holds information about a ray–primitive intersection, including information about the differential geometry of the point on the surface, a pointer to the `Primitive` that the ray hit, and its world-to-object-space transformation. It is defined in the files `core/intersection.h` and `core/intersection.cpp`.

⟨*Intersection Declarations*⟩ ≡
```
struct Intersection {
    ⟨Intersection Public Methods 484⟩
    ⟨Intersection Public Data 186⟩
};
```

⟨*Intersection Public Data*⟩ ≡ 186
```
DifferentialGeometry dg;
const Primitive *primitive;
Transform WorldToObject, ObjectToWorld;
uint32_t shapeId, primitiveId;
float rayEpsilon;
```

It may be necessary to repeatedly refine a primitive until all of the primitives it has returned are themselves intersectable. The `Primitive::FullyRefine()` utility method handles this task. Its implementation is straightforward. It maintains a queue of primitives to be refined (called `todo` in the following code) and invokes the `Primitive::Refine()` method repeatedly on entries in that queue. Intersectable `Primitives` returned by `Primitive::Refine()` are placed in the `refined` array, while nonintersectable ones are placed back on the `todo` list by the `Refine()` routine.

⟨*Primitive Method Definitions*⟩ +≡
```
void
Primitive::FullyRefine(vector<Reference<Primitive> > &refined) const {
    vector<Reference<Primitive> > todo;
    todo.push_back(const_cast<Primitive *>(this));
    while (todo.size()) {
        ⟨Refine last primitive in todo list 187⟩
    }
}
```

DifferentialGeometry 102
Intersection 186
Primitive 185
Primitive::FullyRefine() 186
Primitive::Intersect() 186
Primitive::Refine() 186
Ray 66
Ray::maxt 67
Reference 1011
Shape::Intersect() 111
Transform 76

⟨*Refine last primitive in todo list*⟩ ≡ 186
```
Reference<Primitive> prim = todo.back();
todo.pop_back();
if (prim->CanIntersect())
    refined.push_back(prim);
else
    prim->Refine(todo);
```

In addition to the geometric methods, a `Primitive` object has three methods related to its material properties. The first, `Primitive::GetAreaLight()`, returns a pointer to the `AreaLight` that describes the primitive's emission distribution, if the primitive is itself a light source. If the primitive is not emissive, this method should return `NULL`.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual const AreaLight *GetAreaLight() const = 0;
```

The other two methods return representations of the light-scattering properties of the material at the given point on the surface. The first, `Primitive::GetBSDF()`, returns a `BSDF` object (introduced in Section 9.1) that describes local light-scattering properties at the intersection point. In addition to the differential geometry at the hit point, this method takes the object-to-world-space transformation and a `MemoryArena` to allocate memory for the returned `BSDF`. Section 9.1.1 discusses the use of the `MemoryArena` for `BSDF` memory allocation in more detail.

The second method, `Primitive::GetBSSRDF()`, returns a `BSSRDF`, which describes subsurface scattering inside the primitive—light that enters the surface at points far from where it exits. While subsurface light transport has little effect on the appearance of objects like metal, cloth, or plastic, it is the dominant light-scattering mechanism for biological materials like skin, thick liquids like milk, etc. The `BSSRDF` is used by the subsurface light transport integrator defined in Section 16.5.

⟨*Primitive Interface*⟩ +≡ 185
```
virtual BSDF *GetBSDF(const DifferentialGeometry &dg,
    const Transform &ObjectToWorld, MemoryArena &arena) const = 0;
virtual BSSRDF *GetBSSRDF(const DifferentialGeometry &dg,
    const Transform &ObjectToWorld, MemoryArena &arena) const = 0;
```

AreaLight 623
BSDF 478
BSSRDF 598
DifferentialGeometry 102
GeometricPrimitive 188
MemoryArena 1015
Primitive 185
Primitive::CanIntersect() 186
Primitive::GetAreaLight() 187
Primitive::GetBSDF() 187
Primitive::GetBSSRDF() 187
Primitive::Refine() 186
Reference 1011
Transform 76

4.1.1 GEOMETRIC PRIMITIVES

The `GeometricPrimitive` class represents a single shape (e.g., a sphere) in the scene. One `GeometricPrimitive` is allocated for each shape in the scene description provided by the user. It is implemented in the files `core/primitive.h` and `core/primitive.cpp`.

⟨*GeometricPrimitive Declarations*⟩ ≡
```
class GeometricPrimitive : public Primitive {
public:
    ⟨GeometricPrimitive Public Methods 188⟩
private:
    ⟨GeometricPrimitive Private Data 188⟩
};
```

Each GeometricPrimitive holds a reference to a Shape and its Material. In addition, because primitives in pbrt may be area light sources, it stores a pointer to an AreaLight object that describes its emission characteristics (this pointer is set to NULL if the primitive does not emit light).

⟨*GeometricPrimitive Private Data*⟩ ≡ 187
```
Reference<Shape> shape;
Reference<Material> material;
AreaLight *areaLight;
```

The GeometricPrimitive constructor initializes these variables from the parameters passed to it. Its implementation is omitted.

⟨*GeometricPrimitive Public Methods*⟩ ≡ 187
```
GeometricPrimitive(const Reference<Shape> &s,
                   const Reference<Material> &m, AreaLight *a);
```

Most of the methods of the Primitive interface related to geometric processing are simply forwarded to the corresponding Shape method. For example, GeometricPrimitive::Intersect() calls the Shape::Intersect() method of its enclosed Shape to do the actual geometric intersection and initializes an Intersection object to describe the intersection, if any. It also uses the returned parametric hit distance to update the Ray::maxt member. The primary advantage of storing the distance to the closest hit in Ray::maxt is that this makes it easy to avoid performing intersection tests with any primitives that lie farther along the ray than any already-found intersections.

⟨*GeometricPrimitive Method Definitions*⟩ ≡
```
bool GeometricPrimitive::Intersect(const Ray &r,
                                   Intersection *isect) const {
    float thit, rayEpsilon;
    if (!shape->Intersect(r, &thit, &rayEpsilon, &isect->dg))
        return false;
    isect->primitive = this;
    isect->WorldToObject = *shape->WorldToObject;
    isect->ObjectToWorld = *shape->ObjectToWorld;
    isect->shapeId = shape->shapeId;
    isect->primitiveId = primitiveId;
    isect->rayEpsilon = rayEpsilon;
    r.maxt = thit;
    return true;
}
```

We won't include the implementations of the GeometricPrimitive's WorldBound(), IntersectP(), CanIntersect(), or Refine() methods here; they just forward these requests on to the Shape in a similar manner. Similarly, GetAreaLight() just returns the GeometricPrimitive::areaLight member.

The GetBSDF() method uses the Primitive's Shape to find the shading geometry at the point and forwards the request on to the Material.

AreaLight 623
GeometricPrimitive 188
GeometricPrimitive::areaLight 188
GeometricPrimitive::Intersect() 188
Intersection 186
Intersection::dg 186
Intersection::ObjectToWorld 186
Intersection::primitive 186
Intersection::primitiveId 186
Intersection::rayEpsilon 186
Intersection::shapeId 186
Intersection::WorldToObject 186
Material 483
Primitive 185
Primitive::primitiveId 185
Ray 66
Ray::maxt 67
Reference 1011
Shape 108
Shape::Intersect() 111
Shape::shapeId 109
Shape::WorldToObject 108

⟨*GeometricPrimitive Method Definitions*⟩ +≡

```
BSDF *GeometricPrimitive::GetBSDF(const DifferentialGeometry &dg,
                                  const Transform &ObjectToWorld,
                                  MemoryArena &arena) const {
    DifferentialGeometry dgs;
    shape->GetShadingGeometry(ObjectToWorld, dg, &dgs);
    return material->GetBSDF(dg, dgs, arena);
}
```

The `GeometricPrimitive::GetBSSRDF()` method is similar; it also computes the shading geometry and returns the `BSSRDF` returned by the `Material::GetBSSRDF()` method.

4.1.2 `TransformedPrimitive`: OBJECT INSTANCING AND ANIMATED PRIMITIVES

`TransformedPrimitive` holds a single `Primitive` and also includes an `AnimatedTransform` that is essentially injected in between the underlying primitive and its representation in the scene. This extra transformation enables two useful features: object instancing and primitives with animated transformations.

Object instancing is a classic technique in rendering that reuses transformed copies of a single collection of geometry at multiple positions in a scene. For example, in a model of a concert hall with thousands of identical seats, the scene description can be compressed substantially if all of the seats refer to a shared geometric representation of a single seat. The ecosystem scene in Figure 4.1 has over 4000 individual plants of various types, although only 61 unique plant models. Because each plant model is instanced multiple times with a different transformation for each instance, the complete scene has a total of 19.5 million triangles, although only 1.1 million triangles are stored in memory, thanks to primitive reuse through object instancing. `pbrt` uses approximately 600 MB of memory when rendering this scene with object instancing, but would need upwards of 11 GB to render it without instancing.

Animated transformations enable rigid-body animation of primitives in the scene via the `AnimatedTransform` class. See Figure 2.14 for an image that exhibits motion blur due to animated transformations.

Recall that the `Shapes` of Chapter 3 themselves had object-to-world transformations applied to them to place them in the scene. If a shape is held by a `TransformedPrimitive`, then the shape's notion of world space isn't the actual scene world space—only after the `TransformedPrimitive`'s transformation is also applied is the shape actually in world space. For the applications here, it makes sense for the shape to not be at all aware of the additional transformations being applied. For animated shapes, it's simpler to isolate all of the handling of animated transformations to a single class here, rather than require all `Shapes` to support `AnimatedTransforms`. Similarly, for instanced primitives, letting `Shapes` know all of the instance transforms is of limited utility: we wouldn't want the `TriangleMesh` to make a copy of its vertex positions for each instance transformation and transform them all the way to world space, since this would negate the memory savings of object instancing.

The `TransformedPrimitive` constructor takes a reference to the `Primitive` that represents the model, and the transformation that places it in the scene. If the geometry is described

AnimatedTransform 96
BSDF 478
BSSRDF 598
DifferentialGeometry 102
GeometricPrimitive 188
GeometricPrimitive::material 188
GeometricPrimitive::shape 188
Material::GetBSDF() 483
Material::GetBSSRDF() 484
MemoryArena 1015
Primitive 185
Shape 108
Shape::GetShadingGeometry() 113
Transform 76
TransformedPrimitive 190
TriangleMesh 135

by multiple Primitives, the calling code is responsible for placing them in an Aggregate class so that only a single Primitive needs to be stored here.

The TransformedPrimitive also requires that the primitive be intersectable. (In the presence of object instancing, it would be a waste of both time and memory for all of the instances to individually refine the primitive.) For the code that refines shapes and creates aggregates as needed, see the pbrtObjectInstance() function in Section B.3.6 of Appendix B for the code that creates primitive instances, and see the pbrtShape() function in Section B.3.5 for the corresponding code for animated shapes.

⟨*TransformedPrimitive Declarations*⟩ ≡
```
class TransformedPrimitive : public Primitive {
public:
    ⟨TransformedPrimitive Public Methods 190⟩
private:
    ⟨TransformedPrimitive Private Data 190⟩
};
```

⟨*TransformedPrimitive Public Methods*⟩ ≡ 190
```
TransformedPrimitive(Reference<Primitive> &prim,
                     const AnimatedTransform &w2p)
    : primitive(prim), WorldToPrimitive(w2p) { }
```

⟨*TransformedPrimitive Private Data*⟩ ≡ 190
```
Reference<Primitive> primitive;
const AnimatedTransform WorldToPrimitive;
```

The key task of the TransformedPrimitive is to bridge the Primitive interface that it implements and the Primitive that it holds a reference to, accounting for the effects of the additional transformation matrix that it holds. The TransformedPrimitive's WorldToPrimitive transformation defines the transformation from world space to the coordinate system of this particular instance of the geometry. The primitive member has its own transformation that should be interpreted as the transformation from a TransformedPrimitive's coordinate system to object space. The complete transformation to world space requires both of these transformations together.

Thus, the TransformedPrimitive::Intersect() method transforms the given ray to the primitive's coordinate system and passes the transformed ray to its Intersect() routine. If a hit is found, the maxt value from the transformed ray needs to be copied into the ray r originally passed to the Intersect() routine and the Intersection's primitiveId member is overridden with the primitive id of this TransformedPrimitive.

⟨*TransformedPrimitive Method Definitions*⟩ ≡
```
bool TransformedPrimitive::Intersect(const Ray &r,
                                     Intersection *isect) const {
    Transform w2p;
    WorldToPrimitive.Interpolate(r.time, &w2p);
    Ray ray = w2p(r);
    if (!primitive->Intersect(ray, isect))
        return false;
```

Aggregate 192
AnimatedTransform 96
AnimatedTransform::Interpolate() 99
Intersection 186
Intersection::primitiveId 186
pbrtObjectInstance() 1070
pbrtShape() 1065
Primitive 185
Primitive::Intersect() 186
Primitive::primitiveId 185
Ray 66
Ray::maxt 67
Ray::time 67
Reference 1011
Transform 76
Transform::IsIdentity() 77
Transform::operator() 86
TransformedPrimitive 190
TransformedPrimitive::Intersect() 190
TransformedPrimitive::primitive 190
TransformedPrimitive::WorldToPrimitive 190

```
        r.maxt = ray.maxt;
        isect->primitiveId = primitiveId;
        if (!w2p.IsIdentity()) {
            ⟨Compute world-to-object transformation for instance 191⟩
            ⟨Transform instance's differential geometry to world space 191⟩
        }
        return true;
    }
```

The `Transforms` in the `Intersection` must be set properly as well; we need to compute the full transformation all the way from the primitive's object space to the actual world space, multiplying both of the relevant `Transforms` together.

⟨*Compute world-to-object transformation for instance*⟩ ≡ 190

```
isect->WorldToObject = isect->WorldToObject * w2p;
isect->ObjectToWorld = Inverse(isect->WorldToObject);
```

Finally, the `DifferentialGeometry` at the intersection point needs to be transformed to world space; the `primitive` member will already have transformed the differential geometry information to its notion of world space, so here we only need to apply the effect of the additional transformation held here.

⟨*Transform instance's differential geometry to world space*⟩ ≡ 190

```
Transform PrimitiveToWorld = Inverse(w2p);
isect->dg.p = PrimitiveToWorld(isect->dg.p);
isect->dg.nn = Normalize(PrimitiveToWorld(isect->dg.nn));
isect->dg.dpdu = PrimitiveToWorld(isect->dg.dpdu);
isect->dg.dpdv = PrimitiveToWorld(isect->dg.dpdv);
isect->dg.dndu = PrimitiveToWorld(isect->dg.dndu);
isect->dg.dndv = PrimitiveToWorld(isect->dg.dndv);
```

The rest of the geometric `Primitive` methods are forwarded on to the shared instance, with the results similarly transformed as needed by the `TransformedPrimitive`'s transformation.

⟨*TransformedPrimitive Public Methods*⟩ +≡ 190

```
BBox WorldBound() const {
    return WorldToPrimitive.MotionBounds(primitive->WorldBound(), true);
}
```

The `TransformedPrimitive::GetAreaLight()`, `TransformedPrimitive::GetBSDF()`, and `TransformedPrimitive::GetBSSRDF()` methods should never be called. The corresponding methods in the primitive that the ray actually hit will be called instead. Calling the `TransformedPrimitive` implementations (not shown here) results in a run time error.

AnimatedTransform::
MotionBounds() 100
BBox 70
DifferentialGeometry 102
Intersection 186
Intersection::
WorldToObject 186
Primitive 185
Primitive::WorldBound() 185
Transform 76
Transform::Inverse() 77
TransformedPrimitive 190
TransformedPrimitive::
GetAreaLight() 191
TransformedPrimitive::
GetBSDF() 191
TransformedPrimitive::
GetBSSRDF() 191
TransformedPrimitive::
primitive 190
TransformedPrimitive::
WorldToPrimitive 190
Vector::Normalize() 63

4.2 AGGREGATES

Acceleration structures are one of the components at the heart of any ray tracer. Without algorithms to reduce the number of unnecessary ray intersection tests, tracing a single ray through a scene would take time linear in the number of primitives in the scene, since the

ray would need to be tested against each primitive in turn to find the closest intersection. However, doing so is extremely wasteful in most scenes, since the ray passes nowhere near the vast majority of primitives. The goal of acceleration structures is to allow the quick, simultaneous rejection of *groups* of primitives and also to order the search process so that nearby intersections are likely to be found first and farther away ones can potentially be ignored.

Because ray–object intersections can account for the bulk of execution time in ray tracers, there has been a substantial amount of research into algorithms for ray intersection acceleration. We will not try to explore all of this work here, but refer the interested reader to references in the "Further Reading" section at the end of this chapter and in particular Arvo and Kirk's chapter in *An Introduction to Ray Tracing* (Glassner 1989a), which has a useful taxonomy for classifying different approaches to ray-tracing acceleration.

Broadly speaking, there are two main approaches to this problem: spatial subdivision and object subdivision. Spatial subdivision algorithms decompose 3D space into regions (e.g., by superimposing a grid of axis-aligned boxes on the scene) and record which primitives overlap which regions. In some algorithms, the regions may also be adaptively subdivided based on the number of primitives that overlap them. When a ray intersection needs to be found, the sequence of these regions that the ray passes through is computed and only the primitives in the overlapping regions are tested for intersection.

In contrast, object subdivision is based on progressively breaking the objects in the scene down into smaller sets of constituent objects. For example, a model of a room might be broken down into four walls, a ceiling, and a chair. If a ray doesn't intersect the room's bounding volume, then all of its primitives can be culled. Otherwise, the ray is tested against each of them. If it hits the chair's bounding volume, for example, then it might be tested against each of its legs, the seat, and the back. Otherwise, the chair is culled.

Both of these approaches have been quite successful at solving the general problem of ray intersection computational requirements; there's no fundamental reason to prefer one over the other. The `GridAccel` and `KdTreeAccel` in this chapter are both based on the spatial subdivision approach, and the `BVHAccel` is based on object subdivision.

The `Aggregate` class provides an interface for grouping multiple `Primitive` objects together. Because `Aggregates` themselves implement the `Primitive` interface, no special support is required elsewhere in `pbrt` for intersection acceleration. `Integrators` can be written as if there was just a single `Primitive` in the scene, checking for intersections without needing to be concerned about how they're actually found. Furthermore, by implementing acceleration in this way, it is easy to experiment with new acceleration techniques by simply adding a new `Aggregate` primitive to `pbrt`.

⟨*Aggregate Declarations*⟩ ≡
```
class Aggregate : public Primitive {
public:
    ⟨Aggregate Public Methods⟩
};
```

Like `TransformedPrimitives`, `Aggregate` intersection routines leave the `Intersection::primitive` pointer set to the primitive that the ray actually hit, not the aggregate that

Aggregate 192
BVHAccel 209
GridAccel 196
Integrator 740
Intersection::primitive 186
KdTreeAccel 228
Primitive 185
TransformedPrimitive 190

holds the primitive. Because pbrt uses this pointer to obtain information about the primitive being hit (its reflection and emission properties), the Aggregate::GetAreaLight(), Aggregate::GetBSDF(), and Aggregate::GetBSSRDF() methods should never be called, so the implementations of those methods (not shown here) report a run time error.

4.2.1 RAY–BOX INTERSECTIONS

All of the accelerators in this chapter store a BBox that surrounds all of their primitives. This box can be used to quickly determine if a ray doesn't intersect any of the primitives; if the ray misses the box, it also must miss all of the primitives inside it. Furthermore, some accelerators use the point at which the ray enters the bounding box and the point at which it exits as part of the input to their traversal algorithms. Therefore, we will add a BBox method, BBox::IntersectP(), that checks for a ray–box intersection and returns the two parametric t values of the intersection, if any.

One way to think of bounding boxes is as the intersection of three slabs, where a slab is the region of space between two parallel planes. To intersect a ray against a box, we intersect the ray against each of the box's three slabs in turn. Because the slabs are aligned with the three coordinate axes, a number of optimizations can be made in the ray–slab tests.

The basic ray-bounding box intersection algorithm works as follows: We start with a parametric interval that covers that range of positions t along the ray where we're interested in finding intersections; typically, this is $[0, \infty)$. We will then successively compute the two parametric t positions where the ray intersects each axis-aligned slab. We compute the set intersection of the per-slab intersection interval with the current BBox intersection interval, returning failure if we find that the resulting interval is degenerate. If, after checking all three slabs, the interval is nondegenerate, we have the parametric range of the ray that is inside the box. Figure 4.2 illustrates this process, and Figure 4.3 shows the basic geometry of a ray and a slab.

If the BBox::IntersectP() method returns true, the intersection's parametric range is returned in the optional arguments hitt0 and hitt1. Intersections outside of the

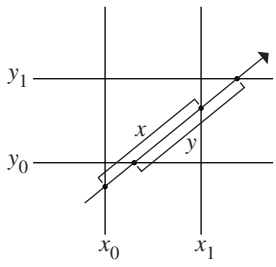


Figure 4.2: Intersecting a Ray with an Axis-Aligned Bounding Box. We compute intersection points with each slab in turn, progressively narrowing the parametric interval. Here, in 2D, the intersection of the x and y extents along the ray gives the extent where the ray is inside the box.

BBox 70

BBox::IntersectP() 194

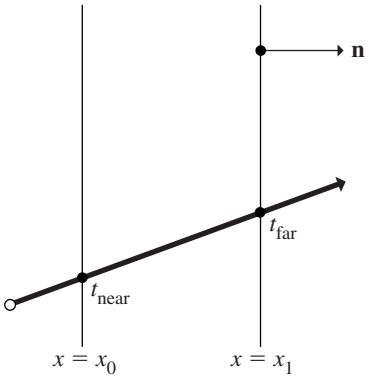


Figure 4.3: Intersecting a Ray with an Axis-Aligned Slab. The two planes shown here are described by $x = c$ for some constant value c. The normal of each plane is $(1, 0, 0)$. Unless the ray is parallel to the planes, it will intersect the slab twice, at parametric positions t_{near} and t_{far}.

[Ray::mint, Ray::maxt] range of the ray are ignored. If the ray's starting point, ray(ray.mint), is inside the box, ray.mint is returned for hit0.

⟨*BBox Method Definitions*⟩ +≡
```
bool BBox::IntersectP(const Ray &ray, float *hit0,
                      float *hit1) const {
    float t0 = ray.mint, t1 = ray.maxt;
    for (int i = 0; i < 3; ++i) {
        ⟨Update interval for ith bounding box slab 195⟩
    }
    if (hit0) *hit0 = t0;
    if (hit1) *hit1 = t1;
    return true;
}
```

For each pair of planes, this routine needs to compute two ray–plane intersections, giving the parametric t values where the intersections occur. Consider the slab along the x axis: it can be described by the two planes through the points $(x_1, 0, 0)$ and $(x_2, 0, 0)$, each with normal $(1, 0, 0)$. Consider the first t value for a plane intersection, t_1. The general form of the intersection between a ray with origin o and direction $\mathbf{d}$ and a plane $ax + by + cz + d = 0$ can be shown to be

$$t = \frac{-d - (\mathrm{o} \cdot (a, b, c))}{(\mathbf{d} \cdot (a, b, c))}.$$

Because the y and z components of the plane's normal are zero, b and c are zero, and a is one. The plane's d coefficient is $-x_1$. We can use this information and the definition of the dot product to simplify this substantially:

$$t_1 = \frac{x_1 - \mathrm{o}_x}{\mathbf{d}_x}.$$

BBox 70
Ray 66
Ray::maxt 67
Ray::mint 67

The code to compute these values starts by computing the reciprocal of the corresponding component of the ray direction so that it can multiply by this factor instead of performing multiple divisions. Note that, although it divides by this component, it is not necessary to verify that it is nonzero. If it is zero, then `invRayDir` will hold an infinite value, either $-\infty$ or ∞, and the rest of the algorithm still works correctly.[1]

⟨*Update interval for* `i` *th bounding box slab*⟩ ≡ 194
```
float invRayDir = 1.f / ray.d[i];
float tNear = (pMin[i] - ray.o[i]) * invRayDir;
float tFar  = (pMax[i] - ray.o[i]) * invRayDir;
⟨Update parametric interval from slab intersection ts 195⟩
```

The two distances are reordered so that `tNear` holds the closer intersection and `tFar` the farther one. This gives a parametric range [`tNear`, `tFar`], which is used to compute the set intersection with the current range [`t0`, `t1`] to compute a new range. If this new range is empty (i.e., `t0` > `t1`), then the code can immediately return failure. There is another floating-point-related subtlety here: in the case where the ray origin is in the plane of one of the bounding box slabs and the ray lies in the plane of the slab, it is possible that `tNear` or `tFar` will be computed by an expression of the form 0/0, which results in an IEEE floating-point "not a number" (NaN) value. Like infinity values, NaNs have well-specified semantics: for example, any logical comparison involving a NaN always evaluates to false. Therefore, the code that updates the values of `t0` and `t1` is carefully written so that if `tNear` or `tFar` is NaN, then `t0` or `t1` won't ever take on a NaN value but will always remain unchanged.

⟨*Update parametric interval from slab intersection ts*⟩ ≡ 195
```
if (tNear > tFar) swap(tNear, tFar);
t0 = tNear > t0 ? tNear : t0;
t1 = tFar  < t1 ? tFar  : t1;
if (t0 > t1) return false;
```

4.3 GRID ACCELERATOR

`GridAccel` is an accelerator that divides an axis-aligned region of space into equal-sized box-shaped chunks (called *voxels*). Each voxel stores references to the primitives that overlap it (Figure 4.4). Given a ray, the grid steps through each of the voxels that the ray passes through in order, checking for intersections with only the primitives in each voxel. Useless ray intersection tests are reduced substantially because primitives far away from the ray aren't considered at all. Furthermore, because the voxels are considered from near to far along the ray, it is possible to stop performing intersection tests once an intersection has been found and it is certain that it is not possible for there to be any closer intersections.

`BBox::pMax` 71
`BBox::pMin` 71
`GridAccel` 196
`Ray::d` 67
`Ray::o` 67

1 This assumes that the architecture being used supports IEEE floating-point arithmetic (Institute of Electrical and Electronic Engineers 1985), which is universal on modern systems. The relevant properties of IEEE floating-point arithmetic are that for all $v > 0$, $v/0 = \infty$ and for all $w < 0$, $w/0 = -\infty$, where ∞ is a special value such that any positive number multiplied by ∞ gives ∞, any negative number multiplied by ∞ gives $-\infty$, and so on.

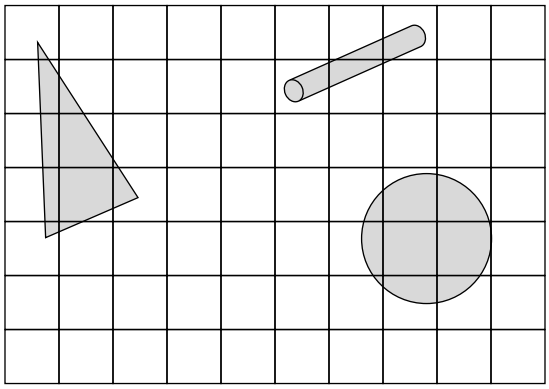


Figure 4.4: The regular grid accelerator divides space into regularly sized cells. Each one stores a reference to the `Primitive`s that overlap it.

The `GridAccel` structure can be initialized quickly, and a simple computation determines the sequence of voxels through which a given ray passes. However, this simplicity is a doubled-edged sword. `GridAccel` can suffer from poor performance when the primitives in the scene aren't distributed evenly throughout space. If there's a small region of space with a lot of geometry in it, all that geometry might fall in a single voxel, and performance will suffer when a ray passes through that voxel, as many intersection tests will be performed. This is sometimes referred to as the "teapot in a stadium" problem; it is not unusual to have such a variable distribution of geometry in realistic scenes.

The root problem is that the data structure cannot adapt well to the distribution of the data that it is storing: if a very fine grid is used, too much time is spent stepping through empty space, and if the grid is too coarse, there is little benefit from the grid at all. The `BVHAccel` and the `KdTreeAccel` in the next two sections adapt to the distribution of geometry such that they don't suffer from this problem.

The implementation of `pbrt`'s grid accelerator is defined in `accelerators/grid.h` and `accelerators/grid.cpp`.

⟨*GridAccel Declarations*⟩ ≡
```
class GridAccel : public Aggregate {
public:
    ⟨GridAccel Public Methods 208⟩
private:
    ⟨GridAccel Private Methods 200⟩
    ⟨GridAccel Private Data 198⟩
};
```

`Aggregate` 192
`BVHAccel` 209
`GridAccel` 196
`KdTreeAccel` 228
`Primitive` 185

4.3.1 CREATION

The `GridAccel` constructor takes a vector of `Primitives` to be stored in the grid. It automatically determines the number of voxels to store in the grid based on the number of primitives.

One factor that adds to the complexity of the grid's implementation is the fact that some of these primitives may not be directly intersectable (they may return `false` from `Primitive::CanIntersect()`) and need to refine themselves into subprimitives before intersection tests can be performed. This is a problem because when the grid is being built we might have a scene with a single primitive in it and choose to build a coarse grid with few voxels. However, if the primitive is later refined for intersection tests, it might turn into millions of primitives, and the original grid resolution would be far too small to efficiently find intersections. `pbrt` addresses this problem in one of two ways:

- If the `refineImmediately` flag to the grid constructor is `true`, all of the `Primitives` are refined until they have turned into intersectable primitives. This may waste time and memory for scenes where some of the primitives wouldn't ever need to be refined since no rays approached them.
- Otherwise, primitives are refined only when a ray enters one of the voxels they are stored in. If they create multiple `Primitives` when refined, the new primitives are stored in a new instance of a `GridAccel` that replaces the original `Primitive` in the top-level grid. This allows the implementation to handle primitive refinement without needing to rebuild the entire grid each time another primitive is refined.

Lazy refinement of primitives in the grid introduces some issues related to multithreaded synchronization (recall the discussion of this topic in Section 1.3.5); the issue is that if one thread is traversing the grid while another thread is modifying its contents by refining primitives in the grid, we need to ensure that one thread's modification of the shared data doesn't cause the other thread to access a partially updated or an otherwise inconsistent representation of the scene. We will discuss these issues (and a solution to them) further later in this section.

⟨*GridAccel Method Definitions*⟩ ≡

```
GridAccel::GridAccel(const vector<Reference<Primitive> > &p,
                     bool refineImmediately) {
    ⟨Initialize primitives with primitives for grid 198⟩
    ⟨Compute bounds and choose grid resolution 198⟩
    ⟨Compute voxel widths and allocate voxels 199⟩
    ⟨Add primitives to grid voxels 199⟩
    ⟨Create reader-writer mutex for grid 205⟩
}
```

GridAccel 196
Primitive 185
Primitive::CanIntersect() 186
Reference 1011

First, the constructor determines the set of `Primitives` to store in the grid, either directly using the primitives passed in or refining all of them until they are intersectable.

⟨*Initialize* primitives *with primitives for grid*⟩ ≡ 197
```
if (refineImmediately)
    for (uint32_t i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(primitives);
else
    primitives = p;
```

⟨*GridAccel Private Data*⟩ ≡ 196
```
vector<Reference<Primitive> > primitives;
```

The constructor next computes the overall bounds of the primitives and determines how many voxels to create along each of the x, y, and z axes. The voxelsPerUnitDist value, computed in a later fragment, gives the average number of voxels that should be created per unit distance in each of the three directions. Given that value, multiplication by the grid's extent in each direction gives the number of voxels to make. The number of voxels in any direction is capped at 64 to avoid creating enormous data structures for complex scenes.

⟨*Compute bounds and choose grid resolution*⟩ ≡ 197
```
for (uint32_t i = 0; i < primitives.size(); ++i)
    bounds = Union(bounds, primitives[i]->WorldBound());
Vector delta = bounds.pMax - bounds.pMin;
⟨Find voxelsPerUnitDist for grid 199⟩
for (int axis = 0; axis < 3; ++axis) {
    nVoxels[axis] = Round2Int(delta[axis] * voxelsPerUnitDist);
    nVoxels[axis] = Clamp(nVoxels[axis], 1, 64);
}
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
int nVoxels[3];
BBox bounds;
```

As a first approximation to choosing a grid size, the total number of voxels should be roughly proportional to the total number of primitives. If the primitives were uniformly distributed, this would mean that a constant number of primitives were in each voxel. While increasing the number of voxels improves efficiency by reducing the average number of primitives per voxel (and thus reducing the number of ray–object intersection tests that need to be performed), doing so also increases memory use, hurts cache performance, and increases the time spent tracing the ray's path through the greater number of voxels it overlaps. On the other hand, too few voxels obviously leads to poor performance, due to an increased number of ray–primitive intersection tests to be performed.

Given the goal of having the number of voxels be proportional to the number of primitives, the cube root of the number of objects is an appropriate starting point for the grid resolution in each direction. In practice, this value is typically scaled by an empirically chosen factor; in pbrt we use a scale of three. Whichever of the x, y, or z dimensions has the largest extent will have exactly $3\sqrt[3]{N}$ voxels for a scene with N primitives. The number of voxels in the other two directions are set in an effort to create voxels that are as close to cubes as possible. The voxelsPerUnitDist variable is the foundation of these

BBox 70
BBox::pMax 71
BBox::pMin 71
Clamp() 1000
GridAccel::nVoxels 198
GridAccel::primitives 198
Primitive 185
Primitive::FullyRefine() 186
Primitive::WorldBound() 185
Reference 1011
Round2Int() 1002
Union() 72
Vector 57

computations; it gives the number of voxels to create per unit distance. Its value is set such that cubeRoot voxels will be created along the axis with the largest extent.

⟨*Find* voxelsPerUnitDist *for grid*⟩ ≡ 198
```
int maxAxis = bounds.MaximumExtent();
float invMaxWidth = 1.f / delta[maxAxis];
float cubeRoot = 3.f * powf(float(primitives.size()), 1.f/3.f);
float voxelsPerUnitDist = cubeRoot * invMaxWidth;
```

Given the number of voxels in each dimension, the constructor sets GridAccel::width, which holds the world space widths of the voxels in each direction. It also precomputes the GridAccel::invWidth values, so that routines that would otherwise divide by the width value can perform a multiplication rather than dividing. Finally, it allocates an array of pointers to Voxel structures for each of the voxels in the grid. These pointers are set to NULL initially and will be allocated only for any voxel with one or more overlapping primitives.[2]

⟨*Compute voxel widths and allocate voxels*⟩ ≡ 197
```
for (int axis = 0; axis < 3; ++axis) {
    width[axis] = delta[axis] / nVoxels[axis];
    invWidth[axis] = (width[axis] == 0.f) ? 0.f : 1.f / width[axis];
}
int nv = nVoxels[0] * nVoxels[1] * nVoxels[2];
voxels = AllocAligned<Voxel *>(nv);
memset(voxels, 0, nv * sizeof(Voxel *));
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
Vector width, invWidth;
Voxel **voxels;
```

Once the voxels themselves have been allocated, primitives can be added to the voxels that they overlap. The GridAccel constructor adds each primitive's corresponding Primitive pointer to the voxels that its bounding box overlaps.

⟨*Add primitives to grid voxels*⟩ ≡ 197
```
for (uint32_t i = 0; i < primitives.size(); ++i) {
    ⟨Find voxel extent of primitive 200⟩
    ⟨Add primitive to overlapping voxels 200⟩
}
```

First, the world space bounds of the primitive are converted to the integer voxel coordinates that contain its two opposite corners. This is done by the utility function GridAccel::posToVoxel(), which turns a world space (x, y, z) position into the coordinates of the voxel that contains that point.

AllocAligned() 1013
BBox::MaximumExtent() 73
GridAccel 196
GridAccel::invWidth 199
GridAccel::nVoxels 198
GridAccel::posToVoxel() 200
GridAccel::width 199
Primitive 185
Vector 57
Voxel 202

2 Some grid implementations try to save even more memory by using a hash table from (x, y, z) voxel number to voxel structures. This saves the memory for the voxels array, which may be substantial if the grid has very small voxels, and the vast majority of them are empty. However, this approach increases the computational expense of finding the Voxel structure for each voxel that a ray passes through.

⟨*Find voxel extent of primitive*⟩ ≡ 199
```
BBox pb = primitives[i]->WorldBound();
int vmin[3], vmax[3];
for (int axis = 0; axis < 3; ++axis) {
    vmin[axis] = posToVoxel(pb.pMin, axis);
    vmax[axis] = posToVoxel(pb.pMax, axis);
}
```

⟨*GridAccel Private Methods*⟩ ≡ 196
```
int posToVoxel(const Point &P, int axis) const {
    int v = Float2Int((P[axis] - bounds.pMin[axis]) *
                      invWidth[axis]);
    return Clamp(v, 0, nVoxels[axis]-1);
}
```

The `GridAccel::voxelToPos()` method is the opposite of `GridAccel::posToVoxel()`; it returns the position of a particular voxel's lower corner.

⟨*GridAccel Private Methods*⟩ +≡ 196
```
float voxelToPos(int p, int axis) const {
    return bounds.pMin[axis] + p * width[axis];
}
```

The primitive is now added to all of the voxels that its bounds overlap. Using its bounds for this test is a conservative test for voxel overlap—at worst it will overestimate the voxels that the primitive overlaps. Figure 4.5 shows an example of two cases where this method leads to primitives being stored in more voxels than necessary. Exercise 4.5 at the end of this chapter describes a more accurate method for associating primitives with voxels.

⟨*Add primitive to overlapping voxels*⟩ ≡ 199
```
for (int z = vmin[2]; z <= vmax[2]; ++z)
    for (int y = vmin[1]; y <= vmax[1]; ++y)
        for (int x = vmin[0]; x <= vmax[0]; ++x) {
            int o = offset(x, y, z);
            if (!voxels[o]) {
                ⟨Allocate new voxel and store primitive in it 201⟩
            }
            else {
                ⟨Add primitive to already-allocated voxel 201⟩
            }
        }
```

The `GridAccel::offset()` utility functions give the offset into the `voxels` array for a particular (x, y, z) voxel. It is the standard indexing scheme in C++ for encoding a multidimensional array in a 1D array. We have localized this computation into a separate function, however, in order to make it easier to experiment with different array layouts, such as blocked schemes for improved cache performance.

BBox 70
BBox::pMax 71
BBox::pMin 71
Clamp() 1000
Float2Int() 1002
GridAccel::invWidth 199
GridAccel::nVoxels 198
GridAccel::offset() 201
GridAccel::posToVoxel() 200
GridAccel::voxels 199
GridAccel::voxelToPos() 200
GridAccel::width 199
Point 63
Primitive::WorldBound() 185

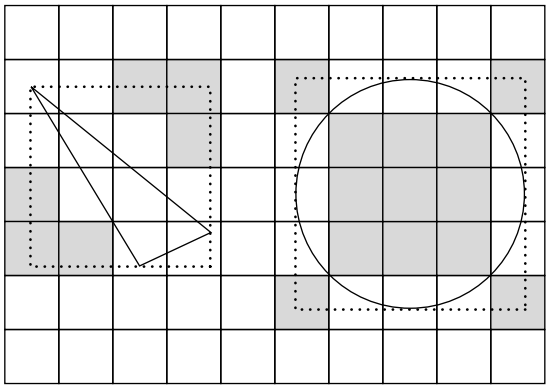


Figure 4.5: Two examples of cases where using the bounding box of a primitive to determine which grid voxels it should be stored in will cause it to be stored in a number of voxels unnecessarily. On the left, a long skinny triangle has a lot of empty space inside its axis-aligned bounding box, and it is unnecessarily added to the shaded voxels. On the right, the surface of the sphere doesn't intersect many of the voxels inside its bound, and they are also inaccurately included in the sphere's extent. While this error degrades performance, it doesn't lead to incorrect ray intersection results.

⟨*GridAccel Private Methods*⟩ +≡ 196
```
inline int offset(int x, int y, int z) const {
    return z*nVoxels[0]*nVoxels[1] + y*nVoxels[0] + x;
}
```

To further reduce memory used for dynamically allocated voxels and to improve their memory locality, the grid constructor uses a `MemoryArena` to hand out memory for voxels. The `MemoryArena`, implemented in Section A.5.4 in Appendix A, provides custom allocation routines based on allocating large blocks of memory and using them to service memory allocation requests. It doesn't support freeing memory from individual allocations; it will only free all of them at once. This improves allocation performance and essentially eliminates memory overhead for bookkeeping, thus reducing the system's overall memory use as well.

⟨*Allocate new voxel and store primitive in it*⟩ ≡ 200
```
voxels[o] = voxelArena.Alloc<Voxel>();
*voxels[o] = Voxel(primitives[i]);
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
MemoryArena voxelArena;
```

GridAccel::nVoxels 198
GridAccel::voxels 199
MemoryArena 1015
Voxel 202
Voxel::AddPrimitive() 202

If this isn't the first primitive to overlap this voxel, the `Voxel` has already been allocated and the primitive is handed off to the `Voxel::AddPrimitive()` method.

⟨*Add primitive to already-allocated voxel*⟩ ≡ 200
```
voxels[o]->AddPrimitive(primitives[i]);
```

The Voxel structure records which primitives overlap its extent using a vector. The Voxel::allCanIntersect member is used to record if all of the primitives in the voxel are intersectable or if some need refinement. It is initially conservatively set to false.

⟨*Voxel Declarations*⟩ ≡
```
struct Voxel {
    ⟨Voxel Public Methods 202⟩
private:
    vector<Reference<Primitive> > primitives;
    bool allCanIntersect;
};
```

When a Voxel is first created, a single Primitive is provided to the constructor.

⟨*Voxel Public Methods*⟩ ≡ 202
```
Voxel(Reference<Primitive> op) {
    allCanIntersect = false;
    primitives.push_back(op);
}
```

⟨*Voxel Public Methods*⟩ +≡ 202
```
void AddPrimitive(Reference<Primitive> prim) {
    primitives.push_back(prim);
}
```

We won't show the straightforward implementations of the GridAccel::WorldBound() or GridAccel::CanIntersect() methods or its destructor.

4.3.2 TRAVERSAL

The GridAccel::Intersect() method handles the task of determining which voxels a ray passes through and calling the appropriate ray–primitive intersection routines.

⟨*GridAccel Method Definitions*⟩ +≡
```
bool GridAccel::Intersect(const Ray &ray, Intersection *isect) const {
    ⟨Check ray against overall grid bounds 203⟩
    ⟨Set up 3D DDA for ray 204⟩
    ⟨Walk ray through voxel grid 205⟩
}
```

The first task is to determine where the ray enters the grid, which gives the starting point for traversal through the voxels. If the ray's origin is inside the grid's bounding box, then clearly it begins there. Otherwise, the GridAccel::Intersect() method finds the intersection of the ray with the grid's bounding box. If the ray hits the bounding box, the first intersection along the ray is the starting point. If the ray misses the grid's bounding box, there can be no intersection with any of the geometry in the grid so GridAccel::Intersect() returns immediately.

GridAccel 196
GridAccel::CanIntersect() 202
GridAccel::Intersect() 202
GridAccel::WorldBound() 202
Intersection 186
Primitive 185
Ray 66
Reference 1011
Voxel 202
Voxel::allCanIntersect 202
Voxel::primitives 202

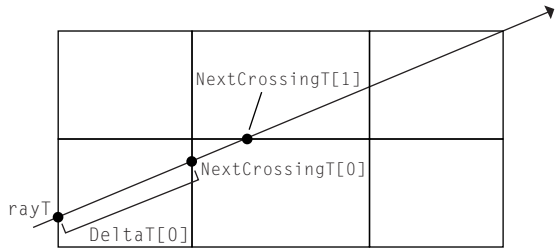


Figure 4.6: Stepping a Ray through a Voxel Grid. `rayT` is the parametric distance along the ray to the first intersection with the grid. The parametric distance along the ray to the point where it crosses into the next voxel in the x direction is stored in `NextCrossingT[0]`, and similarly for the y and z directions (not shown). When the ray crosses into the next x voxel, for example, it is immediately possible to update the value of `NextCrossingT[0]` by adding a fixed value, the voxel width in x divided by the ray's x direction, `DeltaT[0]`.

⟨*Check ray against overall grid bounds*⟩ ≡ 202

```
float rayT;
if (bounds.Inside(ray(ray.mint)))
    rayT = ray.mint;
else if (!bounds.IntersectP(ray, &rayT))
    return false;
Point gridIntersect = ray(rayT);
```

The intersection method next computes the initial (x, y, z) integer voxel coordinates for the ray as well as a number of auxiliary values that will make it efficient to incrementally compute the set of voxels that the ray passes through. The ray–voxel traversal computation is similar in spirit to Bresenham's classic line drawing algorithm, where the series of pixels that a line passes through are found incrementally using just addition and comparisons to step from one pixel to the next. The main difference between the ray marching algorithm and Bresenham's is that we would like to find *all* of the voxels that the ray passes through, while Bresenham's algorithm typically only turns on one pixel per row or column that a line passes through. This type of algorithm is known as a *digital differential analyzer* (DDA).

The values that the ray–voxel stepping algorithm needs to keep track of are the following:

1. The coordinates of the voxel currently being considered, `Pos`.
2. The parametric t position along the ray where it makes its next crossing into another voxel in each of the x, y, and z directions, `NextCrossingT` (Figure 4.6). For example, for a ray with a positive x direction component, the parametric value along the ray where it crosses into the next voxel in x, `NextCrossingT[0]` is the parametric starting point `rayT` plus the x distance to the next voxel divided by the ray's x direction component. (This is similar to the ray–plane intersection formula.)
3. The change in the current voxel coordinates after a step in each direction (1 or −1), stored in `Step`.

BBox::Inside() 72
BBox::IntersectP() 194
Point 63
Ray::mint 67

4. The distance along the ray between voxels in each direction, `DeltaT`. These values are found by dividing the width of a voxel in a particular direction by the ray's corresponding direction component, giving the parametric distance along the ray to travel to get from one side of a voxel to the other in the particular direction.
5. The coordinates of the voxel after the last one the ray passes through when it exits the grid, `Out`.

The first two items will be updated as we step through the grid, while the last three are constant for each ray.

⟨*Set up 3D DDA for ray*⟩ ≡ 202
```
float NextCrossingT[3], DeltaT[3];
int Step[3], Out[3], Pos[3];
for (int axis = 0; axis < 3; ++axis) {
    ⟨Compute current voxel for axis 204⟩
    if (ray.d[axis] >= 0) {
        ⟨Handle ray with positive direction for voxel stepping 204⟩
    }
    else {
        ⟨Handle ray with negative direction for voxel stepping 205⟩
    }
}
```

Computing the voxel address that the ray starts out in is easy since this method has already determined the position where the ray enters the grid. Thus, it can simply use the utility routine `GridAccel::posToVoxel()` defined earlier.

⟨*Compute current voxel for axis*⟩ ≡ 204
```
Pos[axis] = posToVoxel(gridIntersect, axis);
```

If the ray's direction component is zero for a particular axis, then the `NextCrossingT` value for that axis will be initialized to the IEEE floating-point ∞ value by the following computation. The voxel stepping logic later in this section will always decide to step in one of the other directions and will correctly never step in this direction. This is convenient because it can handle rays that are perpendicular to any axis without any special code to test for division by zero.

⟨*Handle ray with positive direction for voxel stepping*⟩ ≡ 204
```
NextCrossingT[axis] = rayT +
    (voxelToPos(Pos[axis]+1, axis) - gridIntersect[axis]) / ray.d[axis];
DeltaT[axis] = width[axis] / ray.d[axis];
Step[axis] = 1;
Out[axis] = nVoxels[axis];
```

Similar computations compute these values for rays with negative direction components:

GridAccel::nVoxels 198
GridAccel::posToVoxel() 200
GridAccel::voxelToPos() 200
GridAccel::width 199
Ray::d 67

⟨*Handle ray with negative direction for voxel stepping*⟩ ≡ 204
```
NextCrossingT[axis] = rayT +
    (voxelToPos(Pos[axis], axis) - gridIntersect[axis]) / ray.d[axis];
DeltaT[axis] = -width[axis] / ray.d[axis];
Step[axis] = -1;
Out[axis] = -1;
```

Once all the preprocessing is done for the ray, stepping through the grid can start. Starting with the first voxel that the ray passes through, the intersection routine checks for intersections with the primitives inside that voxel.

As the ray traverses the grid, it is necessary to handle some issues related to multi-threaded execution. In the `GridAccel` constructor, a reader-writer mutex, `rwMutex`, is created. A reader-writer mutex allows an arbitrary number of threads to request read-only access to shared data. However, if one of the threads wants to modify the data, it must upgrade its hold on the mutex to have write privileges; the `RWMutex` will only allow a single thread to have write privileges and only when no other threads hold read privileges.

⟨*Create reader-writer mutex for grid*⟩ ≡ 197
```
rwMutex = RWMutex::Create();
```

⟨*GridAccel Private Data*⟩ +≡ 196
```
mutable RWMutex *rwMutex;
```

Before traversal starts, a reader lock is acquired from the mutex. If another thread holds a writer lock on the mutex, this thread will stall until the other has released the writer lock, indicating that it has finished updating the grid and it's safe for this thread to continue traversal.

If a hit is found during traversal, the Boolean flag `hitSomething` will be set to `true`. It is necessary to be careful, however, because the found intersection point may be outside the current voxel since primitives may overlap multiple voxels. Therefore, the method doesn't immediately return when done processing a voxel where an intersection was found. Instead, it takes advantage of the fact that the primitive's intersection routine will update the `Ray::maxt` member variable—thus, when stepping through voxels, it will return only when it enters a voxel at a point that is beyond the closest found intersection.

⟨*Walk ray through voxel grid*⟩ ≡ 202
```
RWMutexLock lock(*rwMutex, READ);
bool hitSomething = false;
for (;;) {
    ⟨Check for intersection in current voxel and advance to next 205⟩
}
return hitSomething;
```

⟨*Check for intersection in current voxel and advance to next*⟩ ≡ 205
```
Voxel *voxel = voxels[offset(Pos[0], Pos[1], Pos[2])];
if (voxel != NULL)
    hitSomething |= voxel->Intersect(ray, isect, lock);
⟨Advance to next voxel 207⟩
```

GridAccel 196
GridAccel::offset() 201
GridAccel::rwMutex 205
GridAccel::voxels 199
GridAccel::voxelToPos() 200
GridAccel::width 199
Ray::d 67
Ray::maxt 67
RWMutex 1039
RWMutex::Create() 1039
RWMutexLock 1039
Voxel 202
Voxel::Intersect() 206

For each nonempty voxel, the grid traversal method calls the `Voxel`'s `Intersect()` routine, which handles the details of calling the `Primitive::Intersect()` methods.

⟨*GridAccel Method Definitions*⟩ +≡
```
bool Voxel::Intersect(const Ray &ray, Intersection *isect,
                      RWMutexLock &lock) {
    ⟨Refine primitives in voxel if needed 206⟩
    ⟨Loop over primitives in voxel and find intersections 207⟩
}
```

The Boolean `Voxel::allCanIntersect` member indicates whether all of the primitives in the voxel are known to be intersectable. If its value is `false`, the `Intersect()` routine must loop over all of the primitives, calling their refinement routines as needed until only intersectable geometry remains. If refinement is necessary, we must deal with multi-threaded synchronization: we are going to modify shared data in the grid accelerator, so the read-only lock on the `RWMutex` is upgraded to a writer lock. If other threads are currently holding reader locks, this thread stalls until they have released their reader locks.

Once this thread has a writer lock, it is sure that no other threads are accessing the grid data structures. It is then free to loop over the primitives in the voxel and refine them, modifying the data stored in the voxel. When it is done, it can update `Voxel::allCanIntersect` to be `true`. It then releases its writer lock and continues traversal, holding only a reader lock and thus allowing other threads to access the grid's data.

⟨*Refine primitives in voxel if needed*⟩ ≡ 206
```
if (!allCanIntersect) {
    lock.UpgradeToWrite();
    for (uint32_t i = 0; i < primitives.size(); ++i) {
        Reference<Primitive> &prim = primitives[i];
        ⟨Refine primitive prim if it's not intersectable 207⟩
    }
    allCanIntersect = true;
    lock.DowngradeToRead();
}
```

Primitives that need refinement are refined until only intersectable primitives remain, and a new `GridAccel` is created to hold the returned primitives if more than one was returned. One reason to always make a `GridAccel` for multiple refined primitives is that doing so simplifies primitive refinement. A single `Primitive` always turns into a single object that represents all of the new `Primitives`, so it's never necessary to increase the number of primitives in the voxel. If this primitive overlaps multiple voxels, then because all of them hold a reference to a single `Primitive` for it, it suffices to just update the primitive reference directly, and there's no need to loop over all of the voxels.[3]

GridAccel 196
Intersection 186
Primitive 185
Primitive::Intersect() 186
Ray 66
Reference 1011
RWMutex 1039
RWMutexLock 1039
RWMutexLock::
DowngradeToRead() 1039
RWMutexLock::
UpgradeToWrite() 1039
Voxel 202
Voxel::allCanIntersect 202

3 The bounding box of the original unrefined primitive must encompass the refined geometry as well, so there's no danger that the refined geometry will overlap more voxels than before. On the other hand, it also may overlap many fewer voxels, which would lead to unnecessary intersection tests, since the grid implementation doesn't try to remove references to the primitive from voxels that it no longer overlaps.

⟨*Refine primitive* `prim` *if it's not intersectable*⟩ ≡ 206
```
if (!prim->CanIntersect()) {
    vector<Reference<Primitive> > p;
    prim->FullyRefine(p);
    if (p.size() == 1)
        primitives[i] = p[0];
    else
        primitives[i] = new GridAccel(p, false);
}
```

Once it is certain that there are only intersectable primitives in the voxel, the loop over `Primitives` for performing intersection tests is straightforward.

⟨*Loop over primitives in voxel and find intersections*⟩ ≡ 206
```
bool hitSomething = false;
for (uint32_t i = 0; i < primitives.size(); ++i) {
    Reference<Primitive> &prim = primitives[i];
    if (prim->Intersect(ray, isect))
        hitSomething = true;
}
return hitSomething;
```

After doing the intersection tests for the primitives in the current voxel, it is necessary to step to the next voxel in the ray's path. The grid must decide whether to step in the x, y, or z direction. Fortunately, the `NextCrossingT` variable gives the parametric distance to the next crossing for each direction, and it can choose the smallest one. Traversal can be terminated if this step goes outside of the voxel grid, or if the selected `NextCrossingT` value is beyond the t distance of an already-found intersection. Otherwise, the grid steps to the chosen voxel and increments the chosen direction's `NextCrossingT` by its `DeltaT` value, so that future traversal steps will know how far it is necessary to go before stepping in this direction again.

⟨*Advance to next voxel*⟩ ≡ 205
```
⟨Find stepAxis for stepping to next voxel 208⟩
if (ray.maxt < NextCrossingT[stepAxis])
    break;
Pos[stepAxis] += Step[stepAxis];
if (Pos[stepAxis] == Out[stepAxis])
    break;
NextCrossingT[stepAxis] += DeltaT[stepAxis];
```

Choosing the axis along which to step basically requires finding the smallest of three numbers, a straightforward task. However, in this case an optimization is possible because we don't care about the *value* of the smallest number, just its corresponding index in the `NextCrossingT` array. It is possible to compute this index without any branching, which can lead to performance improvements on modern CPUs, which generally pay a performance penalty for branches.

The following tricky bit of code determines which of the three `NextCrossingT` values is the smallest and sets `stepAxis` accordingly. It encodes this logic by setting each of the

GridAccel 196
Primitive 185
Primitive::
CanIntersect() 186
Primitive::FullyRefine() 186
Primitive::Intersect() 186
Reference 1011

three low-order bits in an integer to the results of three comparisons between pairs of `NextCrossingT` values. It then uses a table (`cmpToAxis`) to map the resulting integer to the direction with the smallest value.

⟨*Find* `stepAxis` *for stepping to next voxel*⟩ ≡ 207
```
int bits = ((NextCrossingT[0] < NextCrossingT[1]) << 2) +
           ((NextCrossingT[0] < NextCrossingT[2]) << 1) +
           ((NextCrossingT[1] < NextCrossingT[2]));
const int cmpToAxis[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };
int stepAxis = cmpToAxis[bits];
```

The grid also provides a special `GridAccel::IntersectP()` method that is optimized for checking for intersection along shadow rays, where we are only interested in the presence of an intersection and not the details of the intersection itself. It is almost identical to the `GridAccel::Intersect()` routine, except that it calls the `Primitive::IntersectP()` method of the primitives rather than `Primitive::Intersect()`, and it immediately stops traversal when any intersection is found. Because of the small number of differences, we won't include the implementation here.

⟨*GridAccel Public Methods*⟩ ≡ 196
```
bool IntersectP(const Ray &ray) const;
```

4.4 BOUNDING VOLUME HIERARCHIES

Bounding volume hierarchies (BVHs) are an approach for ray intersection acceleration based on primitive subdivision, where the primitives are partitioned into a hierarchy of disjoint sets. (In contrast, spatial subdivision generally partitions space into a hierarchy of disjoint sets.) Figure 4.7 shows a bounding volume hierarchy for a simple scene. Primitives are stored in the leaves and each node stores a bounding box of the primitives in the nodes beneath it. Thus, as a ray traverses through the tree, any time it doesn't intersect a node's bounds, the subtree beneath that node can be skipped.

One property of primitive subdivision is that each primitive appears in the hierarchy only once. In contrast, a primitive may overlap many grid voxels, and thus may be tested for intersection multiple times as the ray passes through them.[4] Another implication of this property is that the amount of memory needed to represent the hierarchy is bounded. For a binary BVH that stores a single primitive in each leaf, the total number of nodes is $2n - 1$, where n is the number of primitives. There will be n leaf nodes and $n - 1$ interior nodes. If leaves store multiple primitives, fewer nodes are needed.

BVHs are generally almost as efficient to build as grids are, while delivering much better performance thanks to being able to better adapt to irregular distributions of primitives in the scene. The kd-trees in the following section generally deliver slightly faster ray intersection calculations than BVHs but take substantially longer to build. On the other

GridAccel::Intersect() 202
GridAccel::IntersectP() 208
Primitive::Intersect() 186
Primitive::IntersectP() 186
Ray 66

4 The *mailboxing* technique can be used to avoid multiple intersections for accelerators that use spatial subdivision, though its implementation can be tricky in the presence of multi-threading. More information on mailboxing is available in the "Further Reading" section.

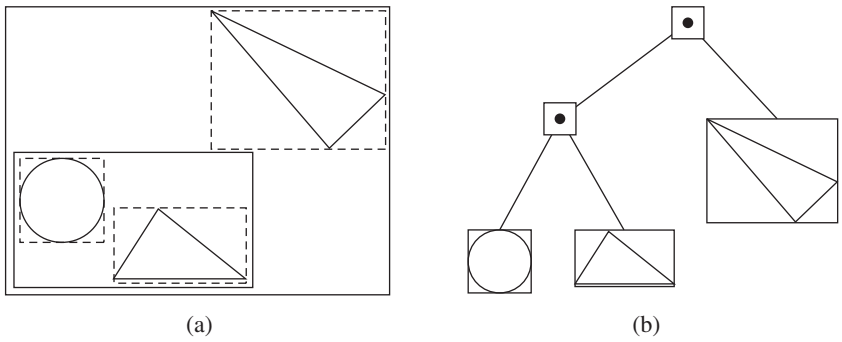


Figure 4.7: Bounding Volume Hierarchy for a Simple Scene. (a) A small collection of primitives, with bounding boxes shown by dashed lines. The primitives are aggregated based on proximity; here, the sphere and the equilateral triangle are bounded by another bounding box before being bounded by a bounding box that encompasses the entire scene (both shown in solid lines). (b) The corresponding bounding volume hierarchy. The root node holds the bounds of the entire scene. Here, it has two children, one storing a bounding box that encompasses the sphere and equilateral triangle (that in turn has those primitives as its children) and the other storing the bounding box that holds the skinny triangle.

hand, BVHs are generally more numerically robust and less prone to subtle round-off bugs than kd-trees are.

The BVH accelerator, `BVHAccel`, is defined in `accelerators/bvh.h` and `accelerators/bvh.cpp`. In addition to the primitives to be stored and the maximum number of primitives that can be in any leaf node, its constructor takes a string that describes which of three algorithms to use when partitioning primitives to build the tree. The default, "sah," indicates that an algorithm based on the "surface area heuristic" (discussed in Section 4.4.2) should be used. The other two approaches take slightly less computation when building the tree but create trees that are typically less efficient when used for ray intersections.

⟨*BVHAccel Method Definitions*⟩ ≡

```
BVHAccel::BVHAccel(const vector<Reference<Primitive> > &p,
                   uint32_t mp, const string &sm) {
    maxPrimsInNode = min(255u, mp);
    for (uint32_t i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(primitives);
    if (sm == "sah")         splitMethod = SPLIT_SAH;
    else if (sm == "middle") splitMethod = SPLIT_MIDDLE;
    else if (sm == "equal")  splitMethod = SPLIT_EQUAL_COUNTS;
    else {
        Warning("BVH split method \"%s\" unknown.  Using \"sah\".",
                sm.c_str());
        splitMethod = SPLIT_SAH;
    }
```

BVHAccel 209
BVHAccel::splitMethod 210
Primitive 185
Primitive::FullyRefine() 186
Reference 1011

```
    if (primitives.size() == 0) {
        nodes = NULL;
        return;
    }
    ⟨Build BVH from primitives 210⟩
}
```

⟨*BVHAccel Private Data*⟩ ≡
```
uint32_t maxPrimsInNode;
enum SplitMethod { SPLIT_MIDDLE, SPLIT_EQUAL_COUNTS, SPLIT_SAH };
SplitMethod splitMethod;
vector<Reference<Primitive> > primitives;
```

4.4.1 BVH CONSTRUCTION

There are three stages to BVH construction. First, bounding information about each primitive is computed and stored in an array that will be used during tree construction. Next, the tree is built via a procedure that splits the primitives into subsets and recursively builds BVHs for the subsets. The result is a binary tree where each interior node holds pointers to its children and each leaf node holds references to one or more primitives. Finally, this tree is converted to a more compact (and thus more efficient) pointerless representation for use during rendering. (The implementation is more straightforward with this approach, versus computing the pointerless representation directly during construction, which is also possible.)

⟨*Build BVH from* primitives⟩ ≡ 209
```
⟨Initialize buildData array for primitives 210⟩
⟨Recursively build BVH tree for primitives 211⟩
⟨Compute representation of depth-first traversal of BVH tree 223⟩
```

For each primitive to be stored in the BVH, we store the centroid of its bounding box, its complete bounding box, and its index in the primitives array in an instance of the BVHPrimitiveInfo structure. As the tree is built, the buildData array will be recursively sorted and partitioned to place the primitives into groups that are spatially close to each other.

⟨*Initialize* buildData *array for primitives*⟩ ≡ 210
```
vector<BVHPrimitiveInfo> buildData;
buildData.reserve(primitives.size());
for (uint32_t i = 0; i < primitives.size(); ++i) {
    BBox bbox = primitives[i]->WorldBound();
    buildData.push_back(BVHPrimitiveInfo(i, bbox));
}
```

BBox 70
BVHAccel::primitives 210
BVHPrimitiveInfo 211
Primitive 185
Primitive::WorldBound() 185
Reference 1011

⟨*BVHAccel Local Declarations*⟩ ≡
```
struct BVHPrimitiveInfo {
    BVHPrimitiveInfo(int pn, const BBox &b)
        : primitiveNumber(pn), bounds(b) {
        centroid = .5f * b.pMin + .5f * b.pMax;
    }
    int primitiveNumber;
    Point centroid;
    BBox bounds;
};
```

The initial call to `recursiveBuild()` is given all of the primitives to be stored in the tree. It returns a pointer to the root of the tree, which is represented with the `BVHBuildNode` structure. The code here uses a `MemoryArena` to allocate nodes one at a time.

One important side-effect of the tree construction process is that a new array of primitives is returned via the `orderedPrims` parameter; this array stores the primitives ordered so that the primitives in leaf nodes occupy contiguous ranges in the array. It is swapped with the original `primitives` array after tree construction.

⟨*Recursively build BVH tree for primitives*⟩ ≡ 210
```
MemoryArena buildArena;
uint32_t totalNodes = 0;
vector<Reference<Primitive> > orderedPrims;
orderedPrims.reserve(primitives.size());
BVHBuildNode *root = recursiveBuild(buildArena, buildData, 0,
                                    primitives.size(), &totalNodes,
                                    orderedPrims);
primitives.swap(orderedPrims);
```

Each `BVHBuildNode` represents a node of the BVH. All nodes store a `BBox`, which stores the bounds of all of the children beneath the node. Each interior node stores pointers to its two children in `children`. Interior nodes also record the coordinate axis along which primitives were sorted for distribution to their two children; this information is used to improve the performance of the traversal algorithm. Leaf nodes need to record which primitive or primitives are stored in them; the elements of the `BVHAccel::primitives` array from the offset `firstPrimOffset` up to but not including `firstPrimOffset + nPrimitives` are the primitives in the leaf. (Hence the need for reordering the primitives array, so that this representation can be used, rather than, for example, storing a variable-sized array of primitive indices at each leaf node.)

⟨*BVHAccel Local Declarations*⟩ +≡
```
struct BVHBuildNode {
    ⟨BVHBuildNode Public Methods 212⟩
    BBox bounds;
    BVHBuildNode *children[2];
    uint32_t splitAxis, firstPrimOffset, nPrimitives;
};
```

BBox 70
BVHAccel::primitives 210
BVHAccel::
recursiveBuild() 213
BVHBuildNode 211
BVHPrimitiveInfo 211
MemoryArena 1015
Point 63
Primitive 185
Reference 1011

The BVHBuildNode constructor only initializes the children pointers; we'll distinguish between leaf and interior nodes by whether their children pointers are NULL or not, respectively.

⟨*BVHBuildNode Public Methods*⟩ ≡ 211
```
BVHBuildNode() { children[0] = children[1] = NULL; }
```

⟨*BVHBuildNode Public Methods*⟩ +≡ 211
```
void InitLeaf(uint32_t first, uint32_t n, const BBox &b) {
    firstPrimOffset = first;
    nPrimitives = n;
    bounds = b;
}
```

The InitInterior() method requires that the two children nodes already have been created, so that their pointers can be passed in. This requirement makes it easy to compute the bounds of the interior node, since the children bounds are immediately available.

⟨*BVHBuildNode Public Methods*⟩ +≡ 211
```
void InitInterior(uint32_t axis, BVHBuildNode *c0, BVHBuildNode *c1) {
    children[0] = c0;
    children[1] = c1;
    bounds = Union(c0->bounds, c1->bounds);
    splitAxis = axis;
    nPrimitives = 0;
}
```

In addition to the MemoryArena used for node allocation and the array of BVHPrimitiveInfo structures, recursiveBuild() takes as parameters the range [start, end). It is responsible for returning a BVH for the subset of primitives represented by the range from buildData[start] up to and including buildData[end-1]. If this represents only a single primitive, the recursion has bottomed out and a leaf node is created. Otherwise, this method partitions the elements of the array in that range using one of a few partitioning algorithms and reorders the array elements in the range accordingly, so that the ranges from [start, mid) and [mid, end) represent the partitioned subsets. If the partitioning is successful, these two primitive sets are in turn passed to recursive calls that will themselves return pointers to nodes for the two children of the current node.

totalNodes tracks the total number of BVH nodes that have been created; this number is used so that exactly the right number of the more compact LinearBVHNodes can be allocated later. Finally, the orderedPrims array is used to store primitive references as primitives are stored in leaf nodes of the tree. This array is initially empty; when a leaf node is created, it adds the primitives that overlap it to the end of the array, making it possible for leaf nodes to just store an offset into this array and a primitive count to represent the set of primitives that overlap it. Recall that when tree construction is finished, BVHAccel::primitives is replaced with the ordered primitives array created here.

BBox 70
BVHAccel::primitives 210
BVHBuildNode 211
BVHBuildNode::bounds 211
BVHBuildNode::children 211
BVHBuildNode::
firstPrimOffset 211
BVHBuildNode::
nPrimitives 211
BVHBuildNode::splitAxis 211
BVHPrimitiveInfo 211
LinearBVHNode 222
MemoryArena 1015
Union() 72

⟨*BVHAccel Method Definitions*⟩ +≡

```
BVHBuildNode *BVHAccel::recursiveBuild(MemoryArena &buildArena,
        vector<BVHPrimitiveInfo> &buildData, uint32_t start,
        uint32_t end, uint32_t *totalNodes,
        vector<Reference<Primitive> > &orderedPrims) {
    (*totalNodes)++;
    BVHBuildNode *node = buildArena.Alloc<BVHBuildNode>();
    ⟨Compute bounds of all primitives in BVH node 213⟩
    uint32_t nPrimitives = end - start;
    if (nPrimitives == 1) {
        ⟨Create leaf BVHBuildNode 213⟩
    }
    else {
        ⟨Compute bound of primitive centroids, choose split dimension dim 214⟩
        ⟨Partition primitives into two sets and build children 215⟩
    }
    return node;
}
```

⟨*Compute bounds of all primitives in BVH node*⟩ ≡ 213

```
BBox bbox;
for (uint32_t i = start; i < end; ++i)
    bbox = Union(bbox, buildData[i].bounds);
```

At leaf nodes, the primitives overlapping the leaf are appended to the `orderedPrims` array and a leaf node object is initialized.

⟨*Create leaf* `BVHBuildNode`⟩ ≡ 213, 215, 221

```
uint32_t firstPrimOffset = orderedPrims.size();
for (uint32_t i = start; i < end; ++i) {
    uint32_t primNum = buildData[i].primitiveNumber;
    orderedPrims.push_back(primitives[primNum]);
}
node->InitLeaf(firstPrimOffset, nPrimitives, bbox);
```

For interior nodes, the collection of primitives must be partitioned between the two children subtrees. Given n primitives, there are in general $2^n - 2$ possible ways to partition them into two nonempty groups. In practice when building BVHs, one generally considers partitions along a coordinate axis, meaning that there are about $6n$ candidate partitions. (Along each axis, each primitive may be put into the first partition or the second partition.)

Here, we choose one of the three coordinate axes to use in partitioning the primitives. We select the axis with the greatest variation of bounding box centroids for the current set of primitives. (An alternative would be to try all three axes and select the one that gave the

BBox 70
BVHBuildNode 211
BVHBuildNode::InitLeaf() 212
BVHPrimitiveInfo 211
BVHPrimitiveInfo::bounds 211
BVHPrimitiveInfo::
primitiveNumber 211
MemoryArena 1015
Primitive 185
Reference 1011

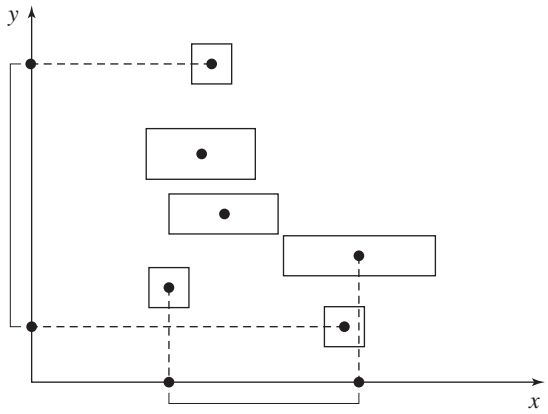


Figure 4.8: Choosing the Axis Along Which to Partition Primitives. The `BVHAccel` chooses an axis along which to partition the primitives based on which axis has the largest range of the centroids of the primitives' bounding boxes. Here, in two dimensions, their extent is largest along the y axis (filled points on the axes), so the primitives will be partitioned in y.

best result, but in practice this approach works well.) This approach gives good partitions in many reasonable scenes; Figure 4.8 illustrates the strategy.

The general goal in partitioning here is to select a partition of primitives that doesn't have too much overlap of the bounding boxes of the two resulting primitive sets—if there is substantial overlap then it will more frequently be necessary to traverse both children subtrees when traversing the tree, requiring more computation than if it had been possible to more effectively prune away collections of primitives. This idea of finding effective primitive partitions will be made more rigorous shortly, in the discussion of the surface area heuristic.

⟨*Compute bound of primitive centroids, choose split dimension* `dim`⟩ ≡ 213

```
BBox centroidBounds;
for (uint32_t i = start; i < end; ++i)
    centroidBounds = Union(centroidBounds, buildData[i].centroid);
int dim = centroidBounds.MaximumExtent();
```

If all of the centroid points are at the same position (i.e., the centroid bounds have zero volume), then recursion stops and a leaf node is created with the primitives; none of the splitting methods here is effective in that (unusual) case. In the usual case, the primitives are partitioned using the chosen method and passed to two recursive calls to `recursiveBuild()`.

BBox 70
BBox::MaximumExtent() 73
BVHPrimitiveInfo::centroid 211
Union() 72

⟨*Partition primitives into two sets and build children*⟩ ≡ 213

```
uint32_t mid = (start + end) / 2;
if (centroidBounds.pMax[dim] == centroidBounds.pMin[dim]) {
    ⟨Create leaf BVHBuildNode 213⟩
    return node;
}
⟨Partition primitives based on splitMethod⟩
node->InitInterior(dim,
                   recursiveBuild(buildArena, buildData, start, mid,
                                  totalNodes, orderedPrims),
                   recursiveBuild(buildArena, buildData, mid, end,
                                  totalNodes, orderedPrims));
```

We won't include the code fragment ⟨*Partition primitives based on* `splitMethod`⟩ here; it just uses the value of `BVHAccel::splitMethod` to determine which primitive partitioning scheme to use. These three schemes will be described in the following few pages.

A simple `splitMethod` is `SPLIT_MIDDLE`, which first computes the midpoint of the primitives' centroids along the splitting axis. This method is implemented in the fragment ⟨*Partition primitives through node's midpoint*⟩. The primitives are classified into the two sets, depending on whether their centroids are above or below the midpoint. This partitioning is easily done with the `std::partition()` C++ standard library function, which takes a range of elements in an array and a comparison function and orders the elements in the array so that all of the elements that return `true` for the given predicate function appear in the range before those that return `false` for it.[5] It returns a pointer to the first element that had a `false` value for the predicate, which is converted into an offset into the `buildData` array so that we can pass it to the recursive call. Figure 4.9 illustrates this method, including cases where it does and does not work well.

⟨*Partition primitives through node's midpoint*⟩ ≡

```
float pmid = .5f * (centroidBounds.pMin[dim] + centroidBounds.pMax[dim]);
BVHPrimitiveInfo *midPtr = std::partition(&buildData[start],
                                          &buildData[end-1]+1,
                                          CompareToMid(dim, pmid));
mid = midPtr - &buildData[0];
```

The `CompareToMid` predicate returns `true` if the given primitive's bound's centroid is below the given midpoint.

`BBox::pMax` 71

`BBox::pMin` 71

`BVHAccel::recursiveBuild()` 213

`BVHAccel::splitMethod` 210

`BVHBuildNode::InitInterior()` 212

`BVHPrimitiveInfo` 211

5 Note the unusual expression of the indexing of the `buildData` array, `&buildData[end-1]+1`. The code is written in this way for somewhat obscure reasons. In the C programming language, it is legal to compute the pointer one element past the end of an array so that iteration over array elements can continue until the current pointer is equal to the end point. To that end, we would like to just write the expression `&buildData[end]` here. However, buildData was allocated as a C++ `vector`; some `vector` implementations issue a run-time error of the offset passed to their `[]` operator is past the end of the array. Because we're not trying to reference the value of the element one past the end of the array but just compute its address, this operation is in fact safe. Therefore, we compute the same address in the end with the expression used here, while also satisfying any `vector` error checking.

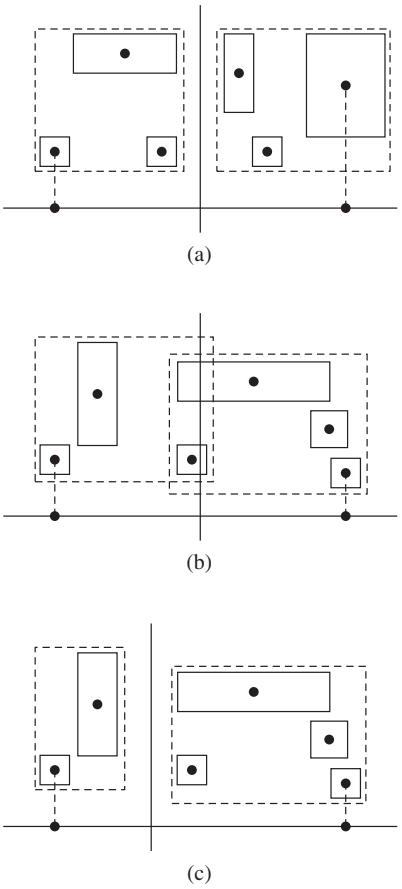


Figure 4.9: Splitting Primitives Based on the Midpoint of Centroids on an Axis. (a) For some distributions of primitives, such as the one shown here, splitting based on the midpoint of the centroids along the chosen axis works well. (The bounding boxes of the two resulting primitive groups are shown with dashed lines.) (b) For distributions like this one, the midpoint is a suboptimal choice; the two resulting bounding boxes overlap substantially. (c) If the same group of primitives from (b) is instead split along the line shown here, the resulting bounding boxes are smaller and don't overlap at all, leading to better performance when rendering.

⟨*BVHAccel Local Declarations*⟩ +≡

```
struct CompareToMid {
    CompareToMid(int d, float m) { dim = d; mid = m; }
    int dim;
    float mid;
    bool operator()(const BVHPrimitiveInfo &a) const {
        return a.centroid[dim] < mid;
    }
};
```

BVHPrimitiveInfo 211
BVHPrimitiveInfo::
centroid 211

Another straightforward partitioning scheme is used when `splitMethod` is `SPLIT_EQUAL_COUNTS`; it is implemented in ⟨*Partition primitives into equally-sized subsets*⟩. It partitions the primitives into two equal-sized subsets such that the first half of the n of them are the $n/2$ with smallest centroid coordinate values along the chosen axis and the second half are the ones with the largest centroid coordinate values. While this approach can sometimes work well, the case in Figure 4.9(b) is one where this method also fares poorly.

This scheme is also easily implemented with a standard library call, `std::nth_element()`. It takes a start, middle, and ending pointer as well as a comparison function. It orders the array so that the element at the middle pointer is the one that would be there if the array was fully sorted, and such that all of the elements before the middle one compare to less than the middle element and all of the elements after it compare to greater than it. This ordering can be done in $O(n)$ time, with n the number of elements, which is more efficient than the $O(n \log n)$ of completely sorting the array.

⟨*Partition primitives into equally-sized subsets*⟩ ≡ 219
```
mid = (start + end) / 2;
std::nth_element(&buildData[start], &buildData[mid],
                 &buildData[end-1]+1, ComparePoints(dim));
```

⟨*BVHAccel Local Declarations*⟩ +≡
```
struct ComparePoints {
    ComparePoints(int d) { dim = d; }
    int dim;
    bool operator()(const BVHPrimitiveInfo &a,
                    const BVHPrimitiveInfo &b) const {
        return a.centroid[dim] < b.centroid[dim];
    }
};
```

4.4.2 THE SURFACE AREA HEURISTIC

The two primitive partitioning approaches above can work well for some distributions of primitives, but they often choose partitions that perform poorly in practice, leading to more nodes of the tree being visited by rays and hence unnecessarily inefficient ray–primitive intersection computations at rendering time. Most of the best current algorithms for building acceleration structures for ray-tracing are based on the "surface area heuristic" (SAH), which provides a well-grounded cost model for answering questions like "which of a number of partitions of primitives will lead to a better BVH for ray–primitive intersection tests?," or "which of a number of possible positions to split space in a spatial subdivision scheme will lead to a better acceleration structure?"

The SAH model estimates the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the time spent on ray–primitive intersection tests for a particular partitioning of primitives. Algorithms for building acceleration structures can then follow the goal of minimizing total cost. Typically, a greedy algorithm is used that minimizes the cost for each single node of the hierarchy being built individually.

BVHPrimitiveInfo 211
BVHPrimitiveInfo::
centroid 211

The ideas behind the SAH cost model are straightforward: at any point in building an adaptive acceleration structure (primitive subdivision or spatial subdivision), we could just create a leaf node for the current region and geometry. In that case, any ray that passes through this region will be tested against all of the overlapping primitives and will incur a cost of

$$\sum_{i=1}^{N} t_{\text{isect}}(i),$$

where N is the number of primitives and $t_{\text{isect}}(i)$ is the time to compute a ray–object intersection with the ith primitive.

The other option is to split the region. In that case, rays will incur the cost

$$c(A, B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i), \tag{4.1}$$

where t_{trav} is the time it takes to traverse the interior node and determine which of the children the ray passes through, p_A and p_B are the probabilities that the ray passes through each of the child nodes (assuming binary subdivision), a_i and b_i are the indices of primitives in the two children nodes, and N_A and N_B are the number of primitives that overlap the regions of the two child nodes, respectively. The choice of how primitives are partitioned affects both the values of the two probabilities as well as the set of primitives on each side of the split.

In `pbrt`, we will make the simplifying assumption that $t_{\text{isect}}(i)$ is the same for all of the primitives; this assumption is probably not too far from reality, and any error that it introduces doesn't seem to affect the performance of accelerators very much. Another possibility would be to add a method to `Primitive` that returns an estimate of the number of CPU cycles its intersection test requires.

The probabilities p_A and p_B can be computed using ideas from geometric probability. It can be shown that for a convex volume A contained in another convex volume B, the conditional probability that a random ray passing through B will also pass through A is the ratio of their surface areas, s_A and s_B:

$$p(A|B) = \frac{s_A}{s_B}.$$

Because we are interested in the cost for rays passing through the node, we can use this result directly. Thus, if we are considering refining a region of space space A such that there are two new subregions with bounds B and C (Figure 4.10), the probability that a ray passing through A will also pass through either of the subregions is easily computed.

When `splitMethod` has the value `SPLIT_SAH`, the SAH is used for building the BVH, choosing a partition of the primitives along the chosen axis that gives a minimal SAH cost estimate by considering a number of candidate partitions. This is the default, and it creates the most efficient trees for rendering. However, once we have refined down to a

Primitive 185

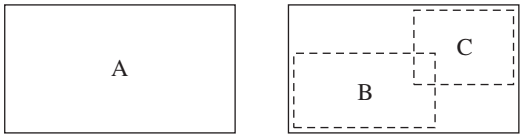


Figure 4.10: If a node of the bounding hierarchy with surface area s_A is split into two children with surface areas s_B and s_C, the probabilities that a ray passing through A also passes through B and C are given by s_B/s_A and s_C/s_A, respectively. Note that $s_B + s_C > s_A$, unless one of them is empty.

small handful of primitives, we switch over to partitioning into equally sized subsets. The incremental computational cost for applying the SAH at this point isn't worthwhile.

⟨*Partition primitives using approximate SAH*⟩ ≡
```
if (nPrimitives <= 4) {
    ⟨Partition primitives into equally-sized subsets 217⟩
}
else {
    ⟨Allocate BucketInfo for SAH partition buckets 219⟩
    ⟨Initialize BucketInfo for SAH partition buckets 220⟩
    ⟨Compute costs for splitting after each bucket 221⟩
    ⟨Find bucket to split at that minimizes SAH metric 221⟩
    ⟨Either create leaf or split primitives at selected SAH bucket 221⟩
}
```

Rather than exhaustively considering all $2n$ possible partitions along the axis, computing the SAH for each to select the best, the implementation here instead divides the range along the axis into a small number of buckets of equal extent. It then only considers partitions at bucket boundaries. This approach is more efficient than considering all partitions while usually still producing partitions that are nearly as effective. This idea is illustrated in Figure 4.11.

⟨*Allocate* BucketInfo *for SAH partition buckets*⟩ ≡ 219
```
const int nBuckets = 12;
struct BucketInfo {
    BucketInfo() { count = 0; }
    int count;
    BBox bounds;
};
BucketInfo buckets[nBuckets];
```

For each primitive in the range, we determine the bucket that its centroid lies in and update the bucket's bounds to include the primitive's bounds.

BBox 70

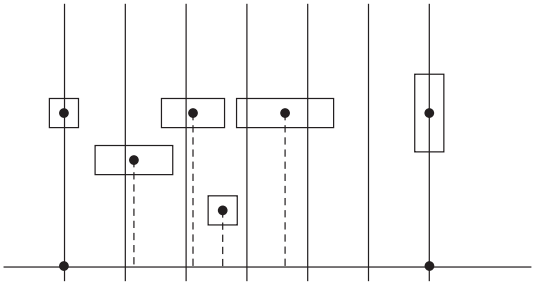


Figure 4.11: Choosing a Splitting Plane with the Surface Area Heuristic for BVHs. The projected extent of primitive bounds centroids is projected onto the chosen split axis. Each primitive is placed in a bucket along the axis based on the centroid of its bounds. The implementation then estimates the cost for splitting the primitives along the planes along each of the bucket boundaries (solid lines); whichever one gives the minimum cost per the surface area heuristic is selected.

⟨*Initialize* `BucketInfo` *for SAH partition buckets*⟩ ≡ 219

```
for (uint32_t i = start; i < end; ++i) {
    int b = nBuckets *
        ((buildData[i].centroid[dim] - centroidBounds.pMin[dim]) /
         (centroidBounds.pMax[dim] - centroidBounds.pMin[dim]));
    if (b == nBuckets) b = nBuckets-1;
    buckets[b].count++;
    buckets[b].bounds = Union(buckets[b].bounds, buildData[i].bounds);
}
```

For each bucket, we now have a count of the number of primitives and the bounds of all of their respective bounding boxes. We want to use the SAH to estimate the cost of splitting at each of the bucket boundaries. The fragment below loops over all of the buckets and initializes the `cost[i]` array to store the estimated SAH cost for splitting after the `i`th bucket. (It doesn't consider a split after the last bucket, which by definition wouldn't split the primitives.)

We arbitrarily set the estimated intersection cost to one, and then set the estimated traversal cost to 1/8. (One of the two of them can always be set to one since it is the relative, rather than absolute, magnitudes of the estimated traversal and intersection costs that determines their effect.) While the absolute amount of computation for node traversal—a ray–bounding box intersection—is only slightly less than the amount of computation needed to intersect a ray with a shape, ray–primitive intersection tests in `pbrt` go through two virtual function calls, which add significant overhead, so we estimate their cost here as eight times more than the ray–box intersection.

This computation has $O(n^2)$ complexity in the number of buckets, though a linear-time implementation based on a forward scan over the buckets and a backward scan over the buckets that incrementally compute and store bounds and counts is possible. For the small n here, the performance impact is generally acceptable, though for a more highly optimized renderer addressing this inefficiency may be worthwhile.

`BucketInfo::bounds` 219
`BucketInfo::count` 219
`BVHPrimitiveInfo::centroid` 211

⟨*Compute costs for splitting after each bucket*⟩ ≡ 219
```
float cost[nBuckets-1];
for (int i = 0; i < nBuckets-1; ++i) {
    BBox b0, b1;
    int count0 = 0, count1 = 0;
    for (int j = 0; j <= i; ++j) {
        b0 = Union(b0, buckets[j].bounds);
        count0 += buckets[j].count;
    }
    for (int j = i+1; j < nBuckets; ++j) {
        b1 = Union(b1, buckets[j].bounds);
        count1 += buckets[j].count;
    }
    cost[i] = .125f + (count0*b0.SurfaceArea() + count1*b1.SurfaceArea()) /
              bbox.SurfaceArea();
}
```

Given all of the costs, a linear scan through the `cost` array finds the partition with minimum cost.

⟨*Find bucket to split at that minimizes SAH metric*⟩ ≡ 219
```
float minCost = cost[0];
uint32_t minCostSplit = 0;
for (int i = 1; i < nBuckets-1; ++i) {
    if (cost[i] < minCost) {
        minCost = cost[i];
        minCostSplit = i;
    }
}
```

If the found bucket boundary for partitioning has a lower estimated cost than building a node with the existing primitives or if more than the maximum number of primitives allowed in a node is present, the `std::partition()` function is used to do the work of reordering nodes in the `buildData` array. Recall from its usage above that this function ensures that all elements of the array that return `true` from the given predicate appear before those that return `false`, and that it returns a pointer to the first element where the predicate returns `false`. Because we arbitrarily set the estimated intersection cost to one previously, the estimated cost for just creating a leaf node is equal to the number of primitives, `nPrimitives`.

⟨*Either create leaf or split primitives at selected SAH bucket*⟩ ≡ 219
```
if (nPrimitives > maxPrimsInNode ||
    minCost < nPrimitives) {
    BVHPrimitiveInfo *pmid = std::partition(&buildData[start],
        &buildData[end-1]+1,
        CompareToBucket(minCostSplit, nBuckets, dim, centroidBounds));
    mid = pmid - &buildData[0];
}
```

BBox 70
BBox::SurfaceArea() 72
BucketInfo::bounds 219
BucketInfo::count 219
BVHAccel::maxPrimsInNode 210
BVHPrimitiveInfo 211
Union() 72

```
else {
    ⟨Create leaf BVHBuildNode 213⟩
}
```

⟨*BVHAccel Local Declarations*⟩ +≡

```
struct CompareToBucket {
    CompareToBucket(int split, int num, int d, const BBox &b)
        : centroidBounds(b)
    { splitBucket = split; nBuckets = num; dim = d; }
    bool operator()(const BVHPrimitiveInfo &p) const;

    int splitBucket, nBuckets, dim;
    const BBox &centroidBounds;
};
```

Classifying a primitive involves recomputing which bucket it maps to and classifying the bucket with respect to the bucket split boundary.

⟨*BVHAccel Local Declarations*⟩ +≡

```
bool CompareToBucket::operator()(const BVHPrimitiveInfo &p) const {
    int b = nBuckets * ((p.centroid[dim] - centroidBounds.pMin[dim]) /
            (centroidBounds.pMax[dim] - centroidBounds.pMin[dim]));
    if (b == nBuckets) b = nBuckets-1;
    return b <= splitBucket;
}
```

4.4.3 COMPACT BVH FOR TRAVERSAL

Once the BVH tree is built, the last step is to convert it into a compact representation—doing so improves cache, memory, and thus overall system performance. The final BVH is stored in a linear array in memory. The nodes of the original tree are laid out in depth-first order, which means that the first child of each interior node is immediately after the node in memory. The offset to the second child of each interior node is then stored explicitly. See Figure 4.12 for an illustration of the relationship between tree topology and node order in memory.

The `LinearBVHNode` structure stores the information needed to traverse the BVH. In addition to the bounding box for each node, for leaf nodes it stores the offset and primitive count for the primitives in the node. For interior nodes, it stores the offset to the second child as well as which of the coordinate axes the primitives were partitioned along when the hierarchy was built; this information is used in the traversal routine below to try to visit nodes in front-to-back order along the ray.

⟨*BVHAccel Local Declarations*⟩ +≡

```
struct LinearBVHNode {
    BBox bounds;
    union {
        uint32_t primitivesOffset;    // leaf
        uint32_t secondChildOffset;   // interior
    };
```

BBox 70
BVHPrimitiveInfo 211
LinearBVHNode 222

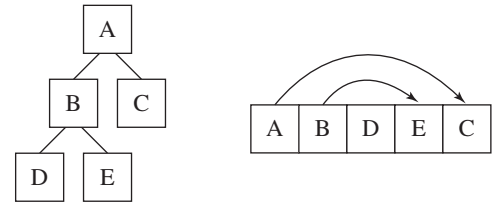


Figure 4.12: Linear Layout of a BVH in Memory. The nodes of the BVH (left) are stored in memory in depth-first order (right). Therefore, for any interior node of the tree (A and B in this example), the first child is found immediately after the parent node in memory. The second child is found via an offset pointer, represented here with lines with arrows. Leaf nodes of the tree (D, E, and C) have no children.

```
    uint8_t nPrimitives;  // 0 -> interior node
    uint8_t axis;         // interior node: xyz
    uint8_t pad[2];       // ensure 32 byte total size
};
```

This structure is padded to ensure that it's 32 bytes large. Doing so ensures that, if the nodes are allocated such that the first node is cache-line aligned, then none of the subsequent nodes will straddle cache lines (as long as the cache line size is at least 32 bytes, which is the case on modern CPU architectures).

⟨*BVHAccel Private Data*⟩ +≡
```
LinearBVHNode *nodes;
```

The built tree is transformed to the `LinearBVHNode` representation by the `flattenBVHTree()` method, which performs a depth-first traversal and stores the nodes in memory in linear order.

⟨*Compute representation of depth-first traversal of BVH tree*⟩ ≡ 210
```
nodes = AllocAligned<LinearBVHNode>(totalNodes);
for (uint32_t i = 0; i < totalNodes; ++i)
    new (&nodes[i]) LinearBVHNode;
uint32_t offset = 0;
flattenBVHTree(root, &offset);
```

BVHAccel::
flattenBVHTree() 224
BVHAccel::nodes 223
LinearBVHNode 222

Flattening the tree to the linear representation is straightforward; the `*offset` parameter tracks the current offset into the `BVHAccel::nodes` array. Note that the current node is added to the array before the recursive calls to process its children (if the node is an interior node).

⟨*BVHAccel Method Definitions*⟩ +≡
```
uint32_t BVHAccel::flattenBVHTree(BVHBuildNode *node, uint32_t *offset) {
    LinearBVHNode *linearNode = &nodes[*offset];
    linearNode->bounds = node->bounds;
    uint32_t myOffset = (*offset)++;
    if (node->nPrimitives > 0) {
        linearNode->primitivesOffset = node->firstPrimOffset;
        linearNode->nPrimitives = node->nPrimitives;
    }
    else {
        ⟨Creater interior flattened BVH node 224⟩
    }
    return myOffset;
}
```

At interior nodes, recursive calls are made to flatten the two subtrees. The first one ends up immediately after the current node in the array, as desired, and the offset of the second one, returned by its recursive `flattenBVHTree()` call, is stored in this node's `secondChildOffset` member.

⟨*Creater interior flattened BVH node*⟩ ≡ 224
```
linearNode->axis = node->splitAxis;
linearNode->nPrimitives = 0;
flattenBVHTree(node->children[0], offset);
linearNode->secondChildOffset = flattenBVHTree(node->children[1],
                                               offset);
```

4.4.4 TRAVERSAL

The BVH traversal code is quite simple—there are no recursive function calls and only a tiny amount of data to maintain about the current state of the traversal. The `Intersect()` method starts by precomputing a few values related to the ray that will be used repeatedly.

⟨*BVHAccel Method Definitions*⟩ +≡
```
bool BVHAccel::Intersect(const Ray &ray, Intersection *isect) const {
    if (!nodes) return false;
    bool hit = false;
    Point origin = ray(ray.mint);
    Vector invDir(1.f / ray.d.x, 1.f / ray.d.y, 1.f / ray.d.z);
    uint32_t dirIsNeg[3] = { invDir.x < 0, invDir.y < 0, invDir.z < 0 };
    ⟨Follow ray through BVH nodes to find primitive intersections 225⟩
    return hit;
}
```

Each time the `while` loop in `Intersect()` starts an iteration, `nodeNum` holds the offset into the `nodes` array of the node to be visited. It starts with a value of zero, representing the root of the tree. The nodes that still need to be visited are stored in the `todo[]` array, which acts as a stack; `todoOffset` holds the offset to the next free element in the stack.

BVHAccel 209
BVHAccel::flattenBVHTree() 224
BVHAccel::nodes 223
BVHBuildNode 211
BVHBuildNode::bounds 211
BVHBuildNode::children 211
BVHBuildNode::firstPrimOffset 211
BVHBuildNode::nPrimitives 211
BVHBuildNode::splitAxis 211
Intersection 186
LinearBVHNode 222
LinearBVHNode::axis 222
LinearBVHNode::bounds 222
LinearBVHNode::nPrimitives 222
LinearBVHNode::primitivesOffset 222
LinearBVHNode::secondChildOffset 222
Point 63
Ray 66
Ray::mint 67
Vector 57

⟨*Follow ray through BVH nodes to find primitive intersections*⟩ ≡ 224
```
uint32_t todoOffset = 0, nodeNum = 0;
uint32_t todo[64];
while (true) {
    const LinearBVHNode *node = &nodes[nodeNum];
    ⟨Check ray against BVH node 225⟩
}
```

At each node, we check to see if the ray intersects the node's bounding box (or starts inside of it). We visit the node if so, testing for intersection with its primitives if it's a leaf node or processing its children if it's an interior node. If no intersection is found, then the offset of the next node to be visited is retrieved from `todo[]` (or, traversal is complete if the stack is empty).

⟨*Check ray against BVH node*⟩ ≡ 225
```
if (::IntersectP(node->bounds, ray, invDir, dirIsNeg)) {
    if (node->nPrimitives > 0) {
        ⟨Intersect ray with primitives in leaf BVH node 226⟩
    }
    else {
        ⟨Put far BVH node on todo stack, advance to near node 227⟩
    }
}
else {
    if (todoOffset == 0) break;
    nodeNum = todo[--todoOffset];
}
```

`BVHAccel` uses a specialized `IntersectP()` function for checking for intersection of rays with bounding boxes. It takes a direction vector that stores the reciprocal of the actual direction, thus changing three divides to multiplies in the slab intersection tests, as well as precomputed values that indicate whether each direction component is negative. The implementation is based on the approach presented by Williams et al. (2005), which performs three main optimizations to the basic `BBox::IntersectP()` routine:

- The `for` loop is unrolled, with the three tests handled directly.
- The reciprocals of the direction components are precomputed and passed in, making it possible to reuse the results of the divisions across all of the bounding box intersection tests done for the ray.
- The sign of the ray's direction components is precomputed, making it possible to eliminate the comparisons of the computed `tNear` and `tFar` values in the original routine and just directly compute the respective near value and the far values. Because the comparisons that order these values from low to high in the original code are dependent on computed values, they can be inefficient for processors to execute, since the computation of their values must be completely finished before the comparison can be made.

`BBox::IntersectP()` 194
`LinearBVHNode` 222

Note also that this routine returns `true` if the ray segment is entirely inside the bounding box, even if the intersections are not within the ray's [`mint`, `maxt`] range; this property is

also desirable for BVH traversal. Because so many ray–bounding box intersection tests are performed while traversing the BVH tree, we found that this optimized method provided approximately a 15% performance improvement in overall rendering time compared to using `BBox::IntersectP()`.

⟨*BVHAccel Local Declarations*⟩ +≡
```
static inline bool IntersectP(const BBox &bounds, const Ray &ray,
        const Vector &invDir, const uint32_t dirIsNeg[3]) {
    ⟨Check for ray intersection against x and y slabs 226⟩
    ⟨Check for ray intersection against z slab⟩
    return (tmin < ray.maxt) && (tmax > ray.mint);
}
```

If the ray direction vector is negative, the "near" parametric intersection will be found with the slab with the larger of the two bounding values, and the far intersection will be found with the slab with the smaller of them. The implementation here uses this observation to compute the near and far parametric values in each direction directly.

⟨*Check for ray intersection against x and y slabs*⟩ ≡ 226
```
float tmin =  (bounds[  dirIsNeg[0]].x - ray.o.x) * invDir.x;
float tmax =  (bounds[1-dirIsNeg[0]].x - ray.o.x) * invDir.x;
float tymin = (bounds[  dirIsNeg[1]].y - ray.o.y) * invDir.y;
float tymax = (bounds[1-dirIsNeg[1]].y - ray.o.y) * invDir.y;
if ((tmin > tymax) || (tymin > tmax))
    return false;
if (tymin > tmin) tmin = tymin;
if (tymax < tmax) tmax = tymax;
```

The fragment ⟨*Check for ray intersection against z slab*⟩ is analogous and isn't included here.

If the current node is a leaf, then the ray must be tested for intersection with the primitives inside it. The next node to visit is then found from the `todo` stack; even if an intersection is found in the current node, the remaining nodes must be visited, in case one of them yields a closer intersection. However, if an intersection is found, the ray's `maxt` value will be updated to the intersection distance; this makes it possible to efficiently discard remaining nodes that are farther away than the intersection.

⟨*Intersect ray with primitives in leaf BVH node*⟩ ≡ 225
```
for (uint32_t i = 0; i < node->nPrimitives; ++i)
    if (primitives[node->primitivesOffset+i]->Intersect(ray, isect))
        hit = true;
if (todoOffset == 0) break;
nodeNum = todo[--todoOffset];
```

For an interior node that the ray hits, it is necessary to visit both of their children. As described above, it's desirable to visit the first child that the ray passes through before visiting the second one, in case there is a primitive that the ray intersects in the first one, so that the ray's `maxt` value can be updated, thus reducing the ray's extent and thus the number of node bounding boxes it intersects.

BBox 70
BBox::IntersectP() 194
BVHAccel::primitives 210
LinearBVHNode::
 nPrimitives 222
Primitive::Intersect() 186
Ray 66
Vector 57

An efficient way to perform a front-to-back traversal without incurring the expense of intersecting the ray with both child nodes and comparing the distances is to use the sign of the ray's direction vector for the coordinate axis along which primitives were partitioned for the current node: if the sign is negative, we should visit the second child before the first child, since the primitives that went into the second child's subtree were on the upper side of the partition point. (And conversely for a positive-signed direction.) Doing this is straightforward: The offset for the node to be visited first is copied to `nodeNum` and the offset for the other node is added to the `todo` stack. (Recall that the first child is immediately after the current node due to the depth-first layout of nodes in memory.)

⟨*Put far BVH node on* `todo` *stack, advance to near node*⟩ ≡ 225

```
if (dirIsNeg[node->axis]) {
   todo[todoOffset++] = nodeNum + 1;
   nodeNum = node->secondChildOffset;
}
else {
   todo[todoOffset++] = node->secondChildOffset;
   nodeNum = nodeNum + 1;
}
```

The `BVHAccel::IntersectP()` method is essentially the same as the regular intersection method, with the usual two differences that `Primitives`' `IntersectP()` methods are called rather than `Intersect()`, and traversal stops immediately if any intersection is found.

4.5 KD-TREE ACCELERATOR

Binary space partitioning (BSP) trees adaptively subdivide space into irregularly sized regions. The most important consequence of this difference with regular grids is that they can be a much more effective data structure for storing irregularly distributed collections of geometry. A BSP tree starts with a bounding box that encompasses the entire scene. If the number of primitives in the box is greater than some threshold, the box is split in half by a plane. Primitives are then associated with whichever half they overlap and primitives that lie in both halves are associated with both of them. (This is in contrast to BVHs, where each primitive is assigned to only one of the two subgroups after a split.)

The splitting process continues recursively either until each leaf region in the resulting tree contains a sufficiently small number of primitives or until a maximum depth is reached. Because the splitting planes can be placed at arbitrary positions inside the overall bound and because different parts of 3D space can be refined to different degrees, BSP trees can easily handle uneven distributions of geometry.

Two variations of BSP trees are *kd-trees* and *octrees*. A kd-tree simply restricts the splitting plane to be perpendicular to one of the coordinate axes; this makes both traversal and construction of the tree more efficient, at the cost of some flexibility in how space is subdivided. The octree uses three axis-perpendicular planes to simultaneously split the box into eight regions at each step (typically by splitting down the center of the extent in each direction). In this section, we will implement a kd-tree for ray intersection

LinearBVHNode::
secondChildOffset 222

Primitive 185

acceleration in the KdTreeAccel class. Source code for this class can be found in the files accelerators/kdtreeaccel.h and accelerators/kdtreeaccel.cpp.

⟨*KdTreeAccel Declarations*⟩ ≡
```
class KdTreeAccel : public Aggregate {
public:
    ⟨KdTreeAccel Public Methods 245⟩
private:
    ⟨KdTreeAccel Private Methods⟩
    ⟨KdTreeAccel Private Data 229⟩
};
```

In addition to the primitives to be stored, the KdTreeAccel constructor takes a few parameters that are used to guide the decisions that will be made as the tree is built; these parameters are stored in member variables (isectCost, traversalCost, maxPrims, maxDepth, and emptyBonus) for later use. For simplicity of implementation, the KdTreeAccel requires that all of the primitives it stores be intersectable. We leave as an exercise the task of improving the implementation to do lazy refinement like the GridAccel does. Therefore, the constructor starts out by refining the given primitives until all are intersectable before building the tree. See Figure 4.13 for an overview of how the tree is built.

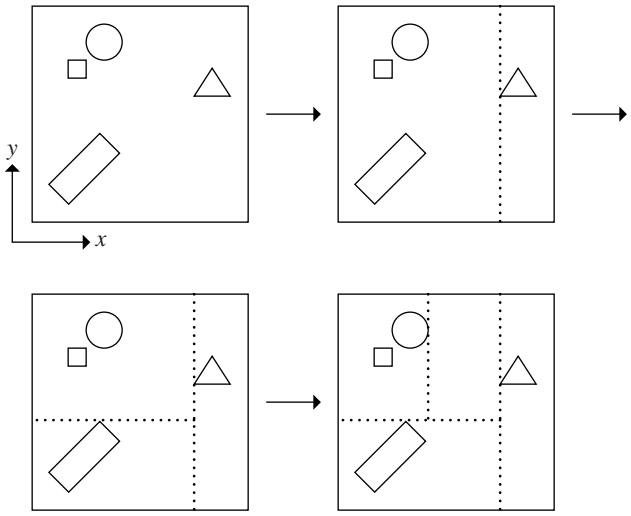


Figure 4.13: The kd-tree is built by recursively splitting the bounding box of the scene geometry along one of the coordinate axes. Here, the first split is along the x axis; it is placed so that the triangle is precisely alone in the right region and the rest of the primitives end up on the left. The left region is then refined a few more times with axis-aligned splitting planes. The details of the refinement criteria—which axis is used to split space at each step, at which position along the axis the plane is placed, and at what point refinement terminates—can all substantially affect the performance of the tree in practice.

Aggregate 192
GridAccel 196
KdTreeAccel 228

⟨*KdTreeAccel Method Definitions*⟩ ≡
```
KdTreeAccel::KdTreeAccel(const vector<Reference<Primitive> > &p,
                         int icost, int tcost, float ebonus, int maxp,
                         int md)
    : isectCost(icost), traversalCost(tcost), maxPrims(maxp), maxDepth(md),
      emptyBonus(ebonus) {
    for (uint32_t i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(primitives);
    ⟨Build kd-tree for accelerator 232⟩
}
```

⟨*KdTreeAccel Private Data*⟩ ≡ 228
```
int isectCost, traversalCost, maxPrims, maxDepth;
float emptyBonus;
vector<Reference<Primitive> > primitives;
```

4.5.1 TREE REPRESENTATION

The kd-tree is a binary tree, where each interior node always has both children and where leaves of the tree store the primitives that overlap them. Each interior node must provide access to three pieces of information:

- Split axis: which of the x, y, or z axes was split at this node.
- Split position: the position of the splitting plane along the axis.
- Children: information about how to reach the two child nodes beneath it.

Each leaf node needs to record only which primitives overlap it.

It is worth going through a bit of trouble to ensure that all interior nodes and many leaf nodes use just 8 bytes of memory (assuming 4-byte `float`s and pointers) because doing so ensures that four nodes will fit into a 32-byte cache line. Because there are often many nodes in the tree and because many nodes are generally accessed for each ray, minimizing the size of the node representation substantially improves cache performance. Our initial implementation used a 16-byte node representation; when we reduced the size to 8 bytes we obtained nearly a 20% speed increase. Both leaves and interior nodes are represented by the following `KdAccelNode` structure. The comments after each `union` member indicate whether a particular field is used for interior nodes, leaf nodes, or both.[6]

⟨*KdTreeAccel Local Declarations*⟩ ≡
```
struct KdAccelNode {
    ⟨KdAccelNode Methods 231⟩
    union {
        float split;            // Interior
        uint32_t onePrimitive;  // Leaf
        uint32_t *primitives;   // Leaf
    };
```

KdAccelNode 229
KdTreeAccel 228
Primitive 185
Primitive::FullyRefine() 186
Reference 1011

6 The attentive reader will note that on a system with 64-bit pointers the `KdAccelNode` structure will actually be 12 bytes, not 8. We leave the correction of this shortcoming to an exercise at the end of the chapter.

```
private:
    union {
        uint32_t flags;         // Both
        uint32_t nPrims;        // Leaf
        uint32_t aboveChild;    // Interior
    };
};
```

The two low-order bits of the `KdAccelNode::flags` variable are used to differentiate between interior nodes with x, y, and z splits (where these bits hold the values 0, 1, and 2, respectively) and leaf nodes (where these bits hold the value 3). It is relatively easy to store leaf nodes in 8 bytes: since the low 2 bits of `KdAccelNode::flags` are used to indicate that this is a leaf, the upper 30 bits of `KdAccelNode::nPrims` are available to record how many primitives overlap it. Then, if just a single primitive overlaps a `KdAccelNode` leaf, an unsigned integer index into the `KdTreeAccel::primitives` array identifies the `Primitive`. If more than one primitive overlaps, memory is dynamically allocated for an array of their indices pointed to by `KdAccelNode::primitives`.

Leaf nodes are easy to initialize, though we have to be careful with the details since both `flags` and `nPrims` share the same storage; we need to be careful to not clobber data for one of them while initializing the other. Furthermore, the number of primitives must be shifted two bits to the left before being stored so that the low two bits of `KdAccelNode::flags` can both be set to 1 to indicate that this is a leaf node.

⟨*KdTreeAccel Method Definitions*⟩ +≡

```
void KdAccelNode::initLeaf(uint32_t *primNums, int np,
                           MemoryArena &arena) {
    flags = 3;
    nPrims |= (np << 2);
    ⟨Store primitive ids for leaf node 230⟩
}
```

For leaf nodes with zero or one overlapping primitives, no dynamic memory allocation is necessary thanks to the `KdAccelNode::onePrimitive` field. For the case where multiple primitives overlap, the caller passes in a `MemoryArena` for allocating memory for the arrays of `Primitive` ids.

⟨*Store primitive ids for leaf node*⟩ ≡ 230

```
if (np == 0)
    onePrimitive = 0;
else if (np == 1)
    onePrimitive = primNums[0];
else {
    primitives = arena.Alloc<uint32_t>(np);
    for (int i = 0; i < np; ++i)
        primitives[i] = primNums[i];
}
```

Getting interior nodes down to 8 bytes is also reasonably straightforward. One 32-bit `float` stores the position along the chosen split axis where the node splits space, and, as

KdAccelNode 229
KdAccelNode::flags 229
KdAccelNode::nPrims 229
KdAccelNode::
onePrimitive 229
KdAccelNode::primitives 229
KdTreeAccel::primitives 229
MemoryArena 1015
MemoryArena::Alloc() 1016
Primitive 185

explained earlier, the lowest 2 bits of `KdAccelNode::flags` are used to record which axis the node was split along. All that is left is to store enough information to find the two children of the node as we're traversing the tree.

Rather than storing two pointers or offsets, we lay the nodes out in a way that lets us only store one child pointer: all of the nodes are allocated in a single contiguous block of memory, and the child of an interior node that is responsible for space below the splitting plane is always stored in the array position immediately after its parent (this also improves cache performance, by keeping at least one child close to its parent in memory). The other child, representing space above the splitting plane, will end up at somewhere else in the array; a single integer offset, `KdAccelNode::aboveChild`, stores its position in the nodes array. This representation is similar to the one used for BVH nodes in Section 4.4.3.

Given all those conventions, the code to initialize an interior node is straightforward. As in the `initLeaf()` method, it's important to assign the value to `flags` before setting `aboveChild`, and to compute the logical OR of the shifted above child value so as to not clobber the bits stored in `flags`.

⟨*KdAccelNode Methods*⟩ ≡ 229
```
void initInterior(uint32_t axis, uint32_t ac, float s) {
    split = s;
    flags = axis;
    aboveChild |= (ac << 2);
}
```

Finally, we'll provide a few methods to extract various values from the node, so that callers don't have to be aware of the subtle details of its representation in memory.

⟨*KdAccelNode Methods*⟩ +≡ 229
```
float SplitPos() const { return split; }
uint32_t nPrimitives() const { return nPrims >> 2; }
uint32_t SplitAxis() const { return flags & 3; }
bool IsLeaf() const { return (flags & 3) == 3; }
uint32_t AboveChild() const { return aboveChild >> 2; }
```

4.5.2 TREE CONSTRUCTION

The kd-tree is built with a recursive top-down algorithm. At each step, we have an axis-aligned region of space and a set of primitives that overlap that region. Either the region is split into two subregions and turned into an interior node, or a leaf node is created with the overlapping primitives, terminating the recursion.

As mentioned in the discussion of `KdAccelNodes`, all tree nodes are stored in a contiguous array. `KdTreeAccel::nextFreeNode` records the next node in this array that is available, and `KdTreeAccel::nAllocedNodes` records the total number that have been allocated. By setting both of them to zero and not allocating any nodes at start-up, the implementation here ensures that an allocation will be done immediately when the first node of the tree is initialized.

KdAccelNode 229
KdAccelNode::aboveChild 229
KdAccelNode::flags 229
KdAccelNode::nPrims 229
KdAccelNode::split 229
KdTreeAccel::
nAllocedNodes 232
KdTreeAccel::
nextFreeNode 232

It is also necessary to determine a maximum tree depth if one wasn't given to the constructor. Although the tree construction process will normally terminate naturally at a reasonable depth, it is important to cap the maximum depth so that the amount of memory used for the tree cannot grow without bound in pathological cases. We have found that the value $8 + 1.3 \log(N)$ gives a reasonable maximum depth for a variety of scenes.

⟨*Build kd-tree for accelerator*⟩ ≡ 229
```
nextFreeNode = nAllocedNodes = 0;
if (maxDepth <= 0)
    maxDepth = Round2Int(8 + 1.3f * Log2Int(float(primitives.size())));
⟨Compute bounds for kd-tree construction 232⟩
⟨Allocate working memory for kd-tree construction 236⟩
⟨Initialize primNums for kd-tree construction 232⟩
⟨Start recursive construction of kd-tree 233⟩
⟨Free working memory for kd-tree construction⟩
```

⟨*KdTreeAccel Private Data*⟩ +≡ 228
```
KdAccelNode *nodes;
int nAllocedNodes, nextFreeNode;
```

Because the construction routine will be repeatedly using the bounding boxes of the primitives along the way, they are stored in a `vector` before tree construction starts so that the potentially slow `Primitive::WorldBound()` methods don't need to be called repeatedly.

⟨*Compute bounds for kd-tree construction*⟩ ≡ 232
```
vector<BBox> primBounds;
primBounds.reserve(primitives.size());
for (uint32_t i = 0; i < primitives.size(); ++i) {
    BBox b = primitives[i]->WorldBound();
    bounds = Union(bounds, b);
    primBounds.push_back(b);
}
```

⟨*KdTreeAccel Private Data*⟩ +≡ 228
```
BBox bounds;
```

One of the parameters to the tree construction routine is an array of primitive indices indicating which primitives overlap the current node. Because all primitives overlap the root node (when the recursion begins) we start with an array initialized with values from zero through `primitives.size()-1`.

⟨*Initialize* `primNums` *for kd-tree construction*⟩ ≡ 232
```
uint32_t *primNums = new uint32_t[primitives.size()];
for (uint32_t i = 0; i < primitives.size(); ++i)
    primNums[i] = i;
```

`KdTreeAccel::buildTree()` is called for each tree node. It is responsible for deciding if the node should be an interior node or leaf and updating the data structures appropriately. The last three parameters, `edges`, `prims0`, and `prims1`, are pointers to data from the

BBox 70
KdAccelNode 229
KdTreeAccel::buildTree() 233
KdTreeAccel::primitives 229
Log2Int() 1001
Primitive::WorldBound() 185
prims0 240
prims1 240
Round2Int() 1002
Union() 72

⟨*Allocate working memory for kd-tree construction*⟩ fragment, which will be defined and documented in a few pages.

⟨*Start recursive construction of kd-tree*⟩ ≡ 232
```
buildTree(0, bounds, primBounds, primNums, primitives.size(),
          maxDepth, edges, prims0, prims1);
```

The main parameters to `KdTreeAccel::buildTree()` are the offset into the array of `KdAccelNodes` to use for the node that it creates, `nodeNum`; the bounding box that gives the region of space that the node covers, `nodeBounds`; and the indices of primitives that overlap it, `primNums`. The remainder of the parameters will be described later, closer to where they are used.

⟨*KdTreeAccel Method Definitions*⟩ +≡
```
void KdTreeAccel::buildTree(int nodeNum, const BBox &nodeBounds,
        const vector<BBox> &allPrimBounds, uint32_t *primNums,
        int nPrimitives, int depth, BoundEdge *edges[3],
        uint32_t *prims0, uint32_t *prims1, int badRefines) {
    ⟨Get next free node from nodes array 233⟩
    ⟨Initialize leaf node if termination criteria met 233⟩
    ⟨Initialize interior node and continue recursion 234⟩
}
```

If all of the allocated nodes have been used up, node memory is reallocated with twice as many entries and the old values are copied. The first time `KdTreeAccel::buildTree()` is called, `KdTreeAccel::nAllocedNodes` is zero and an initial block of tree nodes is allocated.

⟨*Get next free node from* nodes *array*⟩ ≡ 233
```
if (nextFreeNode == nAllocedNodes) {
    int nAlloc = max(2 * nAllocedNodes, 512);
    KdAccelNode *n = AllocAligned<KdAccelNode>(nAlloc);
    if (nAllocedNodes > 0) {
        memcpy(n, nodes, nAllocedNodes * sizeof(KdAccelNode));
        FreeAligned(nodes);
    }
    nodes = n;
    nAllocedNodes = nAlloc;
}
++nextFreeNode;
```

AllocAligned() 1013
BBox 70
BoundEdge 235
KdAccelNode 229
KdAccelNode::initLeaf() 230
KdTreeAccel 228
KdTreeAccel::arena 234
KdTreeAccel::buildTree() 233
KdTreeAccel::maxPrims 229
KdTreeAccel::nAllocedNodes 232
KdTreeAccel::nextFreeNode 232
KdTreeAccel::nodes 232
prims0 240
prims1 240

A leaf node is created (stopping the recursion) either if there are a sufficiently small number of primitives in the region, or if the maximum depth has been reached. The `depth` parameter starts out as the tree's maximum depth and is decremented at each level.

⟨*Initialize leaf node if termination criteria met*⟩ ≡ 233
```
if (nPrimitives <= maxPrims || depth == 0) {
    nodes[nodeNum].initLeaf(primNums, nPrimitives, arena);
    return;
}
```

As described earlier, `KdAccelNode::initLeaf()` uses a memory arena to allocate space for variable-sized arrays of primitives. Because the arena used here is a member variable, all of the memory it allocates will automatically be freed when the `KdTreeAccel` object is destroyed.

⟨*KdTreeAccel Private Data*⟩ +≡ 228
```
MemoryArena arena;
```

If this is an internal node, it is necessary to choose a splitting plane, classify the primitives with respect to that plane, and recurse.

⟨*Initialize interior node and continue recursion*⟩ ≡ 233
⟨*Choose split axis position for interior node* 236⟩
⟨*Create leaf if no good splits were found* 239⟩
⟨*Classify primitives with respect to split* 239⟩
⟨*Recursively initialize children nodes* 240⟩

Our implementation chooses a split using the surface area heuristic (SAH) introduced in Section 4.4.2. The SAH is applicable to kd-trees as well as BVHs; here, the estimated cost is computed for a series of candidate splitting planes in the node, and the split that gives the lowest cost is chosen.

In the implementation here, the intersection cost t_{isect} and the traversal cost t_{trav} can be set by the user; their default values are 80 and 1, respectively. Ultimately, it is the ratio of these two values that determines the behavior of the tree-building algorithm.[7] The greater ratio between these values compared to the values used for BVH construction reflects the fact that visiting a kd-tree node is relatively much less expensive than a BVH node.

One modification to the SAH used for BVH trees is that for kd-trees it is worth giving a slight preference to choosing splits where one of the children has no primitives overlapping it, since rays passing through these regions can immediately advance to the next kd-tree node without any ray–primitive intersection tests. Thus, the revised costs for unsplit and split regions are, respectively,

$$t_{\text{isect}} N, \quad \text{and}$$
$$t_{\text{trav}} + (1 - b_e)(p_B N_B t_{\text{isect}} + p_A N_A t_{\text{isect}}),$$

where b_e is a "bonus" value that is zero unless one of the two regions is completely empty, in which case it takes on a value between zero and one.

Given a way to compute the probabilities for the cost model, the only problem to address is how to generate candidate splitting positions and how to efficiently compute the cost for each candidate. It can be shown that the minimum cost with this model will be attained at a split that is coincident with one of the faces of one of the primitive's bounding

7 Many other implementations of this approach seem to use values for these costs that are much closer together, sometimes even approaching equal values (for example, see Hurley et al. 2002). The values used here gave the best performance for a number of test scenes in `pbrt`. We suspect that this discrepancy is due to the fact that ray–primitive intersection tests in `pbrt` require two virtual function calls and a ray world-to-object-space transformation, in addition to the cost of performing the actual intersection test. Highly optimized ray tracers that only support triangle primitives don't pay any of that additional cost. See Section 18.1.2 for further discussion of this design trade-off.

`KdAccelNode::initLeaf()` 230
`KdTreeAccel` 228
`MemoryArena` 1015

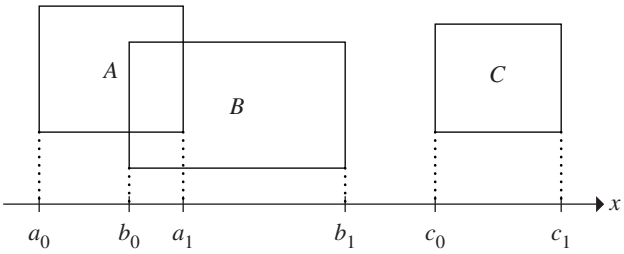


Figure 4.14: Given an axis along which we'd like to consider possible splits, the primitives' bounding boxes are projected onto the axis, which leads to an efficient algorithm to track how many primitives would be on each side of a particular splitting plane. Here, for example, a split at a_1 would leave A completely below the splitting plane, B straddling it, and C completely above it. Each point on the axis, a_0, a_1, b_0, b_1, c_0, and c_1, is represented by an instance of the `BoundEdge` structure.

boxes—there's no need to consider splits at intermediate positions. (To convince yourself of this, consider the behavior of the cost function between the edges of the faces.) Here, we will consider all bounding box faces inside the region for one or more of the three coordinate axes.

The cost for checking all of these candidates thus can be kept relatively low with a carefully structured algorithm. To compute these costs, we will sweep across the projections of the bounding boxes onto each axis and keep track of which gives the lowest cost (Figure 4.14). Each bounding box has two edges on each axis, each of which is represented by an instance of the `BoundEdge` structure. This structure records the position of the edge along the axis, whether it represents the start or end of a bounding box (going from low to high along the axis), and which primitive it is associated with.

⟨*KdTreeAccel Local Declarations*⟩ +≡
```
struct BoundEdge {
    ⟨BoundEdge Public Methods 235⟩
    float t;
    int primNum;
    enum { START, END } type;
};
```

⟨*BoundEdge Public Methods*⟩ ≡ 235
```
BoundEdge(float tt, int pn, bool starting) {
    t = tt;
    primNum = pn;
    type = starting ? START : END;
}
```

BoundEdge 235
BoundEdge::END 235
BoundEdge::primNum 235
BoundEdge::START 235
BoundEdge::t 235
BoundEdge::type 235

At most, `2 * primitives.size()` `BoundEdges` are needed for computing costs for any tree node, so the memory for the edges for all three axes is allocated once and then reused for each node that is created. The fragment ⟨*Free working memory for kd-tree construction*⟩, not included here, frees this space after the tree has been built.

⟨*Allocate working memory for kd-tree construction*⟩ ≡ 232
```
BoundEdge *edges[3];
for (int i = 0; i < 3; ++i)
    edges[i] = new BoundEdge[2*primitives.size()];
```

After determining the estimated cost for creating a leaf, `KdTreeAccel::buildTree()` chooses an axis to try to split along and computes the cost function for each candidate split. `bestAxis` and `bestOffset` record the axis and bounding box edge index that have given the lowest cost so far, `bestCost`. `invTotalSA` is initialized to the reciprocal of the node's surface area; its value will be used when computing the probabilities of rays passing through each of the candidate children nodes.

⟨*Choose split axis position for interior node*⟩ ≡ 234
```
int bestAxis = -1, bestOffset = -1;
float bestCost = INFINITY;
float oldCost = isectCost * float(nPrimitives);
float totalSA = nodeBounds.SurfaceArea();
float invTotalSA = 1.f / totalSA;
Vector d = nodeBounds.pMax - nodeBounds.pMin;
⟨Choose which axis to split along 236⟩
int retries = 0;
retrySplit:
⟨Initialize edges for axis 236⟩
⟨Compute cost of all splits for axis to find best 237⟩
```

This method first tries to find a split along the axis with the largest spatial extent; if successful, this choice helps to give regions of space that tend toward being square in shape. This is an intuitively sensible approach. Later, if it was unsuccessful in finding a good split along this axis, it will go back and try the others in turn.

⟨*Choose which axis to split along*⟩ ≡ 236
```
uint32_t axis = nodeBounds.MaximumExtent();
```

First the `edges` array for the axis is initialized using the bounding boxes of the overlapping primitives. The array is then sorted from low to high along the axis so that it can sweep over the box edges from first to last.

⟨*Initialize edges for* `axis`⟩ ≡ 236
```
for (int i = 0; i < nPrimitives; ++i) {
    int pn = primNums[i];
    const BBox &bbox = allPrimBounds[pn];
    edges[axis][2*i] =   BoundEdge(bbox.pMin[axis], pn, true);
    edges[axis][2*i+1] = BoundEdge(bbox.pMax[axis], pn, false);
}
sort(&edges[axis][0], &edges[axis][2*nPrimitives]);
```

The C++ standard library routine `sort()` requires that the structure being sorted define an ordering; this is done using the `BoundEdge::t` values. However, one subtlety is that if the `BoundEdge::t` values match, it is necessary to try to break the tie by comparing the

BBox 70
BBox::MaximumExtent() 73
BBox::SurfaceArea() 72
BoundEdge 235
BoundEdge::t 235
INFINITY 1002
KdTreeAccel::buildTree() 233
KdTreeAccel::isectCost 229
Vector 57

node's types; this is necessary since `sort()` depends on the fact that the only time `a < b` and `b < a` are both `false` is when `a == b`.

⟨*BoundEdge Public Methods*⟩ +≡ 235
```
bool operator<(const BoundEdge &e) const {
    if (t == e.t)
        return (int)type < (int)e.type;
    else return t < e.t;
}
```

Given the sorted array of edges, we'd like to quickly compute the cost function for a split at each one of them. The probabilities for a ray passing through each child node are easily computed using their surface areas, and the number of primitives on each side of the split is tracked by the variables `nBelow` and `nAbove`. We would like to keep their values updated such that if we chose to split at `edget` for a particular pass through the loop, `nBelow` will give the number of primitives that would end up below the splitting plane and `nAbove` would give the number above it.[8]

At the first edge, all primitives must be above that edge by definition, so `nAbove` is initialized to `nPrimitives` and `nBelow` is set to zero. When the loop is considering a split at the end of a bounding box's extent, `nAbove` needs to be decremented, since that box, which must have previously been above the splitting plane, will no longer be above it if splitting is done at the point. Similarly, after calculating the split cost, if the split candidate was at the start of a bounding box's extent, then the box will be on the below side for all subsequent splits. The tests at the start and end of the loop body update the primitive counts for these two cases.

⟨*Compute cost of all splits for* `axis` *to find best*⟩ ≡ 236
```
int nBelow = 0, nAbove = nPrimitives;
for (int i = 0; i < 2*nPrimitives; ++i) {
    if (edges[axis][i].type == BoundEdge::END) --nAbove;
    float edget = edges[axis][i].t;
    if (edget > nodeBounds.pMin[axis] &&
        edget < nodeBounds.pMax[axis]) {
        ⟨Compute cost for split at ith edge 238⟩
    }
    if (edges[axis][i].type == BoundEdge::START) ++nBelow;
}
```

Given all of this information, the cost for a particular split can be computed. `belowSA` and `aboveSA` hold the surface areas of the two candidate child bounds; they are easily computed by adding up the areas of the six faces.

BBox::pMax 71
BBox::pMin 71
BoundEdge 235
BoundEdge::END 235
BoundEdge::START 235
BoundEdge::t 235
BoundEdge::type 235

8 When multiple bounding box faces project to the same point on the axis, this invariant may not be true at those points. However, as implemented here it will only overestimate the counts and, more importantly, will have the correct value for one of the multiple times through the loop at each of those points, so the algorithm functions correctly in the end anyway.

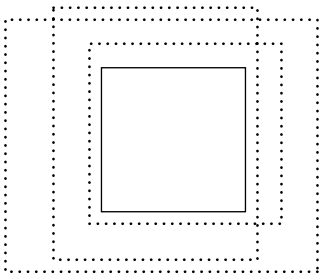


Figure 4.15: If multiple bounding boxes (dotted lines) overlap a kd-tree node (solid lines) as shown here, there is no possible split position that can result in fewer than all of the primitives being on both sides of it.

⟨*Compute cost for split at* `i`*th edge*⟩ ≡ 237

```
uint32_t otherAxis0 = (axis + 1) % 3, otherAxis1 = (axis + 2) % 3;
float belowSA = 2 * (d[otherAxis0] * d[otherAxis1] +
                     (edget - nodeBounds.pMin[axis]) *
                     (d[otherAxis0] + d[otherAxis1]));
float aboveSA = 2 * (d[otherAxis0] * d[otherAxis1] +
                     (nodeBounds.pMax[axis] - edget) *
                     (d[otherAxis0] + d[otherAxis1]));
float pBelow = belowSA * invTotalSA;
float pAbove = aboveSA * invTotalSA;
float eb = (nAbove == 0 || nBelow == 0) ? emptyBonus : 0.f;
float cost = traversalCost +
             isectCost * (1.f - eb) * (pBelow * nBelow + pAbove * nAbove);
```

⟨*Update best split if this is lowest cost so far* 238⟩

If the cost computed for this candidate split is the best one so far, the details of the split are recorded.

⟨*Update best split if this is lowest cost so far*⟩ ≡ 238

```
if (cost < bestCost)  {
    bestCost = cost;
    bestAxis = axis;
    bestOffset = i;
}
```

It may happen that there are no possible splits found in the previous tests (Figure 4.15 illustrates a case where this may happen). In this case, there isn't a single candidate position at which to split the node along the current axis. At this point, splitting is tried for the other two axes in turn. If neither of them can find a split (when `retries` is equal to two), then there is no useful way to refine the node, since both children will still have the same number of overlapping primitives. When this condition occurs, all that can be done is to give up and make a leaf node.

BBox::pMax 71
BBox::pMin 71

It is also possible that the best split will have a cost that is still higher than the cost for not splitting the node at all. If it is substantially worse and there aren't too many primitives, a leaf node is made immediately. Otherwise, `badRefines` keeps track of how many bad splits have been made so far above the current node of the tree. It's worth allowing a few slightly poor refinements since later splits may be able to find better ones given a smaller subset of primitives to consider.

⟨*Create leaf if no good splits were found*⟩ ≡ 234

```
if (bestAxis == -1 && retries < 2) {
    ++retries;
    axis = (axis+1) % 3;
    goto retrySplit;
}
if (bestCost > oldCost) ++badRefines;
if ((bestCost > 4.f * oldCost && nPrimitives < 16) ||
    bestAxis == -1 || badRefines == 3) {
    nodes[nodeNum].initLeaf(primNums, nPrimitives, arena);
    return;
}
```

Having chosen a split position, the bounding box edges can be used to classify the primitives as being above, below, or on both sides of the split in the same way as was done to keep track of `nBelow` and `nAbove` in the earlier code. Note that the `bestOffset` entry in the arrays is skipped in the loops below; this is necessary so that the primitive whose bounding box edge was used for the split isn't incorrectly categorized as being on both sides of the split.

⟨*Classify primitives with respect to split*⟩ ≡ 234

```
int n0 = 0, n1 = 0;
for (int i = 0; i < bestOffset; ++i)
    if (edges[bestAxis][i].type == BoundEdge::START)
        prims0[n0++] = edges[bestAxis][i].primNum;
for (int i = bestOffset+1; i < 2*nPrimitives; ++i)
    if (edges[bestAxis][i].type == BoundEdge::END)
        prims1[n1++] = edges[bestAxis][i].primNum;
```

BoundEdge 235
BoundEdge::END 235
BoundEdge::primNum 235
BoundEdge::START 235
BoundEdge::type 235
KdAccelNode::initLeaf() 230
KdTreeAccel::buildTree() 233
prims0 240
prims1 240

Recall that the node number of the "below" child of this node in the kd-tree nodes array is the current node number plus one. After the recursion has returned from that side of the tree, the `nextFreeNode` offset is used for the "above" child. The only other important detail here is that the `prims0` memory is passed directly for reuse by both children, while the `prims1` pointer is advanced forward first. This is necessary since the current invocation of `KdTreeAccel::buildTree()` depends on its `prims1` values being preserved over the first recursive call to `KdTreeAccel::buildTree()` in the following, since it must be passed as a parameter to the second call. However, there is no corresponding need to preserve the `edges` values or to preserve `prims0` beyond its immediate use in the first recursive call.

⟨*Recursively initialize children nodes*⟩ ≡ 234
```
float tsplit = edges[bestAxis][bestOffset].t;
BBox bounds0 = nodeBounds, bounds1 = nodeBounds;
bounds0.pMax[bestAxis] = bounds1.pMin[bestAxis] = tsplit;
buildTree(nodeNum+1, bounds0,
          allPrimBounds, prims0, n0, depth-1, edges,
          prims0, prims1 + nPrimitives, badRefines);
uint32_t aboveChild = nextFreeNode;
nodes[nodeNum].initInterior(bestAxis, aboveChild, tsplit);
buildTree(aboveChild, bounds1, allPrimBounds, prims1, n1,
          depth-1, edges, prims0, prims1 + nPrimitives, badRefines);
```

Thus, much more space is needed for the `prims1` array of integers for storing the worst-case possible number of overlapping primitive numbers than for the `prims0` array, which only needs to handle the primitives at a single level at a time.

⟨*Allocate working memory for kd-tree construction*⟩ +≡ 232
```
uint32_t *prims0 = new uint32_t[primitives.size()];
uint32_t *prims1 = new uint32_t[(maxDepth+1) * primitives.size()];
```

4.5.3 TRAVERSAL

Figure 4.16 shows the basic process of ray traversal through the tree. Intersecting the ray with the tree's overall bounds gives initial `tmin` and `tmax` values, marked with points in the figure. As with the other accelerators in this chapter, if the ray misses the scene bounds, this method can immediately return `false`. Otherwise, it starts to descend into the tree, starting at the root. At each interior node, it determines which of the two children the ray enters first and processes both children in order. Traversal ends either when the ray exits the tree or when the closest intersection is found.

⟨*KdTreeAccel Method Definitions*⟩ +≡
```
bool KdTreeAccel::Intersect(const Ray &ray,
                            Intersection *isect) const {
    ⟨Compute initial parametric range of ray inside kd-tree extent 240⟩
    ⟨Prepare to traverse kd-tree for ray 241⟩
    ⟨Traverse kd-tree nodes in order for ray 242⟩
}
```

The algorithm starts by finding the overall parametric range $[t_{\min}, t_{\max}]$ of the ray's overlap with the tree, exiting immediately if there is no overlap.

⟨*Compute initial parametric range of ray inside kd-tree extent*⟩ ≡ 240
```
float tmin, tmax;
if (!bounds.IntersectP(ray, &tmin, &tmax))
    return false;
```

The array of `KdToDo` structures is used to record the nodes yet to be processed for the ray; it is ordered so that the last active entry in the array is the next node that should be considered. The maximum number of entries needed in this array is the maximum

BBox 70
BBox::IntersectP() 194
BBox::pMax 71
BBox::pMin 71
BoundEdge::t 235
Intersection 186
KdAccelNode::initInterior() 231
KdTreeAccel 228
KdTreeAccel::buildTree() 233
KdTreeAccel::nextFreeNode 232
KdTreeAccel::primitives 229
prims0 240
prims1 240
Ray 66

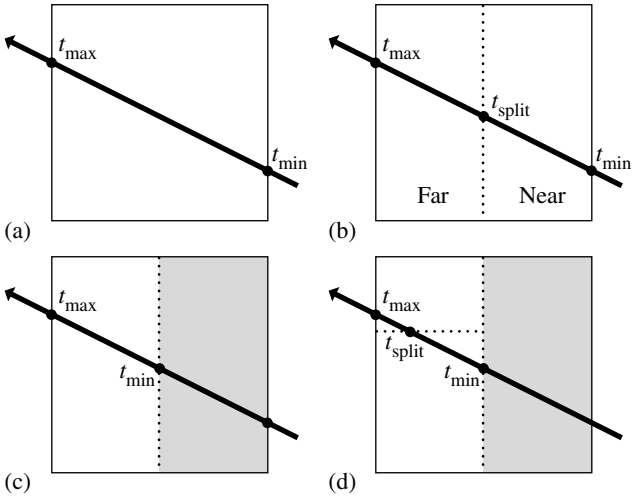


Figure 4.16: Traversal of a Ray through the Kd-Tree. (a) The ray is intersected with the bounds of the tree, giving an initial parametric $[t_{min}, t_{max}]$ range to consider. (b) Because this range is nonempty, it is necessary to consider the two children of the root node here. The ray first enters the child on the right, labeled "near," where it has a parametric range $[t_{min}, t_{split}]$. If the near node is a leaf with primitives in it, ray–primitive intersection tests are performed; otherwise, its children nodes are processed. (c) If no hit is found in the node, or if a hit is found beyond $[t_{min}, t_{split}]$, then the far node, on the left, is processed. (d) This sequence continues—processing tree nodes in a depth-first, front-to-back traversal—until the closest intersection is found or the ray exits the tree.

depth of the kd-tree; the array size used in the following should be more than enough in practice.

⟨*Prepare to traverse kd-tree for ray*⟩ ≡ 240
```
Vector invDir(1.f/ray.d.x, 1.f/ray.d.y, 1.f/ray.d.z);
#define MAX_TODO 64
KdToDo todo[MAX_TODO];
int todoPos = 0;
```

⟨*KdTreeAccel Declarations*⟩ +≡
```
struct KdToDo {
    const KdAccelNode *node;
    float tmin, tmax;
};
```

KdAccelNode 229
KdToDo 241
Ray::d 67
Vector 57

The traversal continues through the nodes, processing a single leaf or interior node each time through the loop. The values `tmin` and `tmax` will always hold the parametric range for the ray's overlap with the current node.

⟨*Traverse kd-tree nodes in order for ray*⟩ ≡ 240
```
bool hit = false;
const KdAccelNode *node = &nodes[0];
while (node != NULL) {
    ⟨Bail out if we found a hit closer than the current node 242⟩
    if (!node->IsLeaf()) {
        ⟨Process kd-tree interior node 242⟩
    }
    else {
        ⟨Check for intersections inside leaf node 244⟩
        ⟨Grab next node to process from todo list 245⟩
    }
}
return hit;
```

An intersection may have been previously found in a primitive that overlaps multiple nodes. If the intersection was outside the current node when first detected, it is necessary to keep traversing the tree until we come to a node where `tmin` is beyond the intersection. Only then is it certain that there is no closer intersection with some other primitive.

⟨*Bail out if we found a hit closer than the current node*⟩ ≡ 242
```
if (ray.maxt < tmin) break;
```

For interior tree nodes the first thing to do is to intersect the ray with the node's splitting plane; given the intersection point, we can determine if one or both of the children nodes need to be processed and in what order the ray passes through them.

⟨*Process kd-tree interior node*⟩ ≡ 242
```
⟨Compute parametric distance along ray to split plane 242⟩
⟨Get node children pointers for ray 243⟩
⟨Advance to next child node, possibly enqueue other child 244⟩
```

The parametric distance to the split plane is computed in the same manner as was done in computing the intersection of a ray and an axis-aligned plane for the ray–bounding box test. We use the precomputed `invDir` value to save a divide each time through the loop.

⟨*Compute parametric distance along ray to split plane*⟩ ≡ 242
```
int axis = node->SplitAxis();
float tplane = (node->SplitPos() - ray.o[axis]) * invDir[axis];
```

Now it is necessary to determine the order in which the ray encounters the children nodes, so that the tree is traversed in front-to-back order along the ray. Figure 4.17 shows the geometry of this computation. The position of the ray's origin with respect to the splitting plane is enough to distinguish between the two cases, ignoring for now the case where the ray doesn't actually pass through one of the two nodes. The rare case when the ray's origin lies on the splitting plane requires careful handling in this case, as its direction needs to be used instead to discriminate between the two cases.

KdAccelNode 229
KdAccelNode::IsLeaf() 231
KdAccelNode::SplitAxis() 231
KdAccelNode::SplitPos() 231
KdTreeAccel::nodes 232
Ray::maxt 67
Ray::o 67

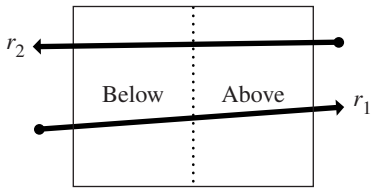


Figure 4.17: The position of the origin of the ray with respect to the splitting plane can be used to determine which of the node's children should be processed first. If the origin of a ray like r_1 is on the "below" side of the splitting plane, we should process the below child before the above child, and vice versa.

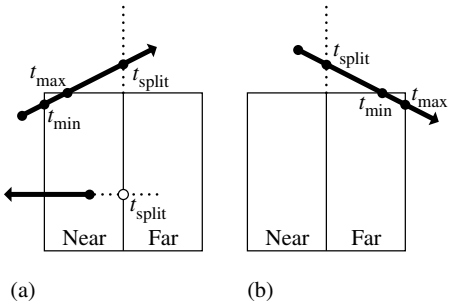


Figure 4.18: Two cases where both children of a node don't need to be processed because the ray doesn't overlap them. (a) The top ray intersects the splitting plane beyond the ray's t_{max} position and thus doesn't enter the far child. The bottom ray is facing away from the splitting plane, indicated by a negative t_{split} value. (b) The ray intersects the plane before the ray's t_{min} value, indicating that the near child doesn't need processing.

⟨*Get node children pointers for ray*⟩ ≡ 242

```
const KdAccelNode *firstChild, *secondChild;
int belowFirst = (ray.o[axis] <  node->SplitPos()) ||
                 (ray.o[axis] == node->SplitPos() && ray.d[axis] >= 0);
if (belowFirst) {
    firstChild = node + 1;
    secondChild = &nodes[node->AboveChild()];
}
else {
    firstChild = &nodes[node->AboveChild()];
    secondChild = node + 1;
}
```

KdAccelNode 229
KdAccelNode::SplitPos() 231
Ray::o 67

It may not be necessary to process both children of this node. Figure 4.18 shows some configurations where the ray only passes through one of the children. The ray will never

miss both children, since otherwise the current interior node should never have been traversed.

The first `if` test in the following code corresponds to Figure 4.18(a): only the near node needs to be processed if it can be shown that the ray doesn't overlap the far node because it faces away from it or doesn't overlap it because $t_{\text{split}} > t_{\text{max}}$. Figure 4.18(b) shows the similar case tested in the second `if` test: the near node may not need processing if the ray doesn't overlap it. Otherwise, the `else` clause handles the case of both children needing processing; the near node will be processed next, and the far node goes on the `todo` list.

⟨*Advance to next child node, possibly enqueue other child*⟩ ≡ 242
```
if (tplane > tmax || tplane <= 0)
    node = firstChild;
else if (tplane < tmin)
    node = secondChild;
else {
    ⟨Enqueue secondChild in todo list 244⟩
    node = firstChild;
    tmax = tplane;
}
```

⟨*Enqueue* `secondChild` *in todo list*⟩ ≡ 244
```
todo[todoPos].node = secondChild;
todo[todoPos].tmin = tplane;
todo[todoPos].tmax = tmax;
++todoPos;
```

If the current node is a leaf, intersection tests are performed against the primitives in the leaf.

⟨*Check for intersections inside leaf node*⟩ ≡ 242
```
uint32_t nPrimitives = node->nPrimitives();
if (nPrimitives == 1) {
    const Reference<Primitive> &prim = primitives[node->onePrimitive];
    ⟨Check one primitive inside leaf node 244⟩
}
else {
    uint32_t *prims = node->primitives;
    for (uint32_t i = 0; i < nPrimitives; ++i) {
        const Reference<Primitive> &prim = primitives[prims[i]];
        ⟨Check one primitive inside leaf node 244⟩
    }
}
```

Processing an individual primitive is just a matter of passing the intersection request on to the primitive.

⟨*Check one primitive inside leaf node*⟩ ≡ 244
```
if (prim->Intersect(ray, isect))
    hit = true;
```

KdAccelNode::nPrimitives() 231
KdAccelNode::onePrimitive 229
KdAccelNode::primitives 229
KdToDo::node 241
KdToDo::tmax 241
KdToDo::tmin 241
KdTreeAccel::primitives 229
Primitive 185
Primitive::Intersect() 186
Reference 1011

After doing the intersection tests at the leaf node, the next node to process is loaded from the `todo` array. If no more nodes remain, then the ray has passed through the tree without hitting anything.

⟨*Grab next node to process from todo list*⟩ ≡ 242
```
if (todoPos > 0) {
    --todoPos;
    node = todo[todoPos].node;
    tmin = todo[todoPos].tmin;
    tmax = todo[todoPos].tmax;
}
else
    break;
```

Like the `GridAccel` and `BVHAccel`, the `KdTreeAccel` has a specialized intersection method for shadow rays that is not shown here. It is similar to the `KdTreeAccel::Intersect()` method, just calling `Primitive::IntersectP()` method and returning `true` as soon as it finds any intersection without worrying about finding the closest one.

⟨*KdTreeAccel Public Methods*⟩ ≡ 228
```
bool IntersectP(const Ray &ray) const;
```

4.6 DEBUGGING AGGREGATES

Bugs in aggregates can be notoriously difficult to find and fix; once an aggregate implementation mostly works, the difficulty is that it does the correct thing for *almost* all of the rays that it's given and it's only a very small subset where a bug manifests itself. Even worse, the bug may be due to a small error made back when the acceleration structure was first built. Working backward from a ray with an incorrect intersection to the original source of the bug can be a very tedious process. We have learned some effective techniques and built some useful testing infrastructure for debugging aggregates that we will discuss here.

First, we need to define what a correct result for an intersection calculation is. We will say that given a ray and a collection of primitives, the correct intersection result is a primitive the ray hits with minimum t value along the ray, subject to the ray's parametric `mint`-`maxt` range, so long as the ray also intersects the bounding box returned by the primitive.

There are a few subtleties in this statement. One is that it is possible that multiple primitives will report an intersection at the same t value. In this case, we say that a correct answer is an intersection with any of those primitives. It's not worth the implementation complexity to define a more specific requirement—for example, that the primitive that appeared first in the input file must be reported; this constraint would make accelerator implementations needlessly complex and introduce overhead that's not generally worthwhile for rendering.

The second subtlety stems from small numeric inconsistencies between the primitive's bounding box and the primitive's intersection routine. There may be rays that are reported to not intersect the bounding box, yet the primitive's `Intersect()` method may

BVHAccel 209
GridAccel 196
KdToDo::node 241
KdToDo::tmax 241
KdToDo::tmin 241
KdTreeAccel 228
KdTreeAccel::Intersect() 240
Primitive::IntersectP() 186
Ray 66

report intersection. Even if the bounding box provided by the primitive is correct (i.e., it fully encompasses the primitive's spatial extent), it is possible that the inevitable small errors from floating-point roundoff in the `BBox::IntersectP()` routine will mean that a ray is reported to not intersect the bounding box even though the primitive's intersection method reports that the ray does in fact intersect it.[9] It's also not reasonable to require accelerators to report only primitive intersections where the ray also intersects the primitive's bounding box: for example, the grid accelerator tests the ray against all of the primitives in a grid voxel, regardless of whether it intersects their bounding boxes; for that algorithm, there isn't any reason to also test the ray against all of the primitives' bounding boxes given the knowledge that the primitive's bound overlapped the grid voxel and the ray passes through the grid voxel.

Given all this, for the purposes of defining a correct accelerator, we will therefore also say that primitive intersections where the ray is not found to intersect the bounding box are not expected to be found by the accelerator, but that it is also not incorrect for an accelerator to report such an intersection.

This definition of correctness for intersection computations means that valid image differences can occur when a scene is rendered with different accelerators. For example, one may choose one instance of an ambiguous intersection, where two or more primitives report an intersection with the same t value, and the other accelerator may report another instance. Even if this were not the case, testing and debugging accelerators by comparing image differences is somewhat unwieldy; given a small difference in two rendered images, working backward to find a bug in an accelerator that led to that difference can be a significant debugging chore.

4.6.1 FINDING BUGS IN AGGREGATES

`pbrt` provides an `AggregateTest` `Renderer` for testing aggregate implementations. Its implementation is straightforward: given a scene, it generates a large number of random rays in the scene and first traces each one using whichever accelerator was specified in the scene's description file. It then exhaustively tests the ray for intersection against every primitive in the scene. If the results are inconsistent (subject to the definition of a correct intersection calculation above), then the accelerator being tested must have a bug. Information about these rays is printed for use in later debugging runs. `AggregateTest` is found in the files `renderers/aggregatetest.h` and `renderers/aggregatetest.cpp`.

Finding and fixing bugs using directed tests like these is generally much easier than finding and fixing them after seeing a surprising error in an image. Isolating a single ray where the accelerator is not computing the right result helps narrow the debugging problem; to the extent that automated tests like those here can find instances of bugs, the easier the debugging process is. Another significant advantage of targeted testing code is that, when one makes changes to the system, one can run the tests and ensure that a subtle bug hasn't been introduced by the tests; the returns from the work to implement tests like these in the first place are generally worthwhile.

AggregateTest 247
BBox::IntersectP() 194
Renderer 24

9 This shortcoming presumably could be addressed through careful analysis of the roundoff error in the `BBox::IntersectP()` routine and by appropriate modifications to it to ensure that the computation is sufficiently conservative that this problem doesn't occur.

The AggregateTest constructor is not included here; it fully refines all of the primitives passed to it, computes their bounding boxes, and stores a count of the number of test iterations to run.

⟨*AggregateTest Private Data*⟩ ≡
```
int nIterations;
vector<Reference<Primitive> > primitives;
vector<BBox> bboxes;
```

Testing is done in the Render() method. It first computes a bounding box that is moderately larger than the full scene extent; random rays will be generated inside this bounding box. It then runs for the number of requested iterations, generating random rays and computing intersections with them.

⟨*AggregateTest Method Definitions*⟩ ≡
```
void AggregateTest::Render(const Scene *scene) {
    RNG rng;
    ⟨Compute bounding box of region used to generate random rays 247⟩
    Point lastHit;
    float lastEps = 0.f;
    for (int i = 0; i < nIterations; ++i) {
        ⟨Choose random rays, rayAccel and rayAll for testing 247⟩
        ⟨Compute intersections using accelerator and exhaustive testing 248⟩
        ⟨Report any inconsistencies between intersections 249⟩
    }
}
```

⟨*Compute bounding box of region used to generate random rays*⟩ ≡ 247
```
BBox bbox = scene->WorldBound();
bbox.Expand(bbox.pMax[bbox.MaximumExtent()] -
            bbox.pMin[bbox.MaximumExtent()]);
```

The ⟨*Choose random rays, rayAccel and rayAll for testing*⟩ generates random rays in the scene. Its goal is to be as efficient as possible at generating rays that are likely to be troublesome and expose corner cases in the implementations of accelerators.[10]

⟨*Choose random rays, rayAccel and rayAll for testing*⟩ ≡ 247
```
⟨Choose ray origin for testing accelerator 248⟩
⟨Choose ray direction for testing accelerator 248⟩
⟨Choose ray epsilon for testing accelerator 248⟩
Ray rayAccel(org, dir, eps);
Ray rayAll = rayAccel;
```

The ray origin is chosen in one of two ways: either as a random point inside the scene's bounding box or at the surface hit by the previous ray. Starting some of the rays on (or, strictly speaking, near) scene surfaces is important; not only are the majority of

AggregateTest::
nIterations 247
BBox 70
BBox::Expand() 72
Point 63
Primitive 185
Ray 66
Reference 1011
RNG 1003
Scene 22
Scene::WorldBound() 24

10 Another effective approach for generating these testing rays would be to log all of the rays generated during the regular process of rendering the scene and then rerun them through this testing code to make sure the intersection results are consistent. This modification is left for an exercise at the end of the chapter.

rays traced in the process of rendering rays leaving intersected surfaces, but errors from incorrect intersection computations from these rays aren't always easily visible. If rays from the camera have egregiously wrong intersection results, the bug will be obviously visible in the rendered image. Rays reflected from surfaces that have wrong intersection results may not have as obvious a visual manifestation.

⟨*Choose ray origin for testing accelerator*⟩ ≡ 247
```
Point org(Lerp(rng.RandomFloat(), bbox.pMin.x, bbox.pMax.x),
          Lerp(rng.RandomFloat(), bbox.pMin.y, bbox.pMax.y),
          Lerp(rng.RandomFloat(), bbox.pMin.z, bbox.pMax.z));
if ((rng.RandomUInt() % 4) == 0) org = lastHit;
```

The ray direction is usually chosen by randomly selecting a direction. However, occasionally setting two of the direction vectors to zero is worthwhile: rays parallel to a coordinate axis can be problematic, so exercising this case is useful.

⟨*Choose ray direction for testing accelerator*⟩ ≡ 247
```
Vector dir = UniformSampleSphere(rng.RandomFloat(), rng.RandomFloat());
if ((rng.RandomUInt() % 32) == 0)      dir.x = dir.y = 0.f;
else if ((rng.RandomUInt() % 32) == 0) dir.x = dir.z = 0.f;
else if ((rng.RandomUInt() % 32) == 0) dir.y = dir.z = 0.f;
```

Finally, the "epsilon" value for the ray (the minimum parameteric distance before which intersections are ignored) is chosen. The implementation randomly chooses between three typical values—zero, the epsilon value returned at the last surface intersection, and a small floating-point value.

⟨*Choose ray epsilon for testing accelerator*⟩ ≡ 247
```
float eps = 0.f;
if (rng.RandomFloat() < .25) eps = lastEps;
else if (rng.RandomFloat() < .25) eps = 1e-3f;
```

Given the ray, `AggregateTest` uses both the regular `Aggregate` and an exhaustive test of all primitives in the scene to check for intersections. Note that it checks for the case where the ray is not reported to intersect the bounding box but still hits the geometry here, setting `inconsistentBounds` in that case. Inconsistent intersections for these rays won't be reported.

⟨*Compute intersections using accelerator and exhaustive testing*⟩ ≡ 247
```
Intersection isectAccel, isectAll;
bool hitAccel = scene->Intersect(rayAccel, &isectAccel);
bool hitAll = false;
bool inconsistentBounds = false;
for (uint32_t j = 0; j < primitives.size(); ++j) {
    if (bboxes[j].IntersectP(rayAll))
        hitAll |= primitives[j]->Intersect(rayAll, &isectAll);
    else if (primitives[j]->Intersect(rayAll, &isectAll))
        inconsistentBounds = true;
}
```

Aggregate 192
BBox::IntersectP() 194
BBox::pMax 71
BBox::pMin 71
Intersection 186
Lerp() 1000
Point 63
Primitive::Intersect() 186
RNG::RandomFloat() 1003
RNG::RandomUInt() 1003
Scene::Intersect() 23
Vector 57

As long as no intersections were found where the ray hit the primitive but not its bounding box, then if the exhaustive test and the accelerator compute different parametric t values for the intersection point, a warning is printed here, including information about the origin and direction of the ray that exhibits the bug. Rather than use the conventional `%f` `printf()` formatting for the floating-point values, the code here uses `%a`, which prints the given value as a hexidecimal floating-point value. For example, the value 2.5 is printed `0x1.4p+1`. This representation of floating-point values is useful in that it is guaranteed to represent the floating-point value exactly, without any roundoff error. (In contrast, the number will generally be rounded when using `%f`.) It's important that the ray that exhibits the bug be stored precisely, since a slight perturbation to it may not exhibit the bug any more.

⟨*Report any inconsistencies between intersections*⟩ ≡ 247

```
if (!inconsistentBounds &&
    ((hitAccel != hitAll) || (rayAccel.maxt != rayAll.maxt)))
    Warning("Disagreement: t accel %.16g [%a] t exhaustive %.16g [%a]\n"
            "Ray: org [%a, %a, %a], dir [%a, %a, %a], mint = %a",
            rayAccel.maxt, rayAll.maxt, rayAccel.maxt, rayAll.maxt,
            rayAll.o.x, rayAll.o.y, rayAll.o.z,
            rayAll.d.x, rayAll.d.y, rayAll.d.z, rayAll.mint);
if (hitAll) {
    lastHit = rayAll(rayAll.maxt);
    lastEps = isectAll.rayEpsilon;
}
```

4.6.2 FIXING BUGS IN AGGREGATES

Finding a ray, scene, and accelerator where an incorrect intersection is found is only a start; tracking down the actual bug from this point is not easy. The first step in fixing this sort of bug is to determine which primitive is the one that should have been determined to be the closest intersecting primitive—from there, the question is "Why wasn't the ray found to intersect with it?" Adding code to the accelerator's constructor to immediately trace the ray that was found to hit the bug after the acceleration data structure is built gives an easy point at which to set a breakpoint in the debugger.

For accelerators based on spatial subdivision, one can take the position of the missed intersection and determine which spatial region the intersection point lies in. (The intersection point may also lie on the boundary between two nodes, which can be a problematic case for implementations.) The bug then must come from one of two causes:

- There is an error in the traversal code, and the ray never passes through the node where the intersection lies.
- Or, there is an error in the code that builds the accelerator, and the primitive is not present in the node where the intersection occurs.

`Intersection::rayEpsilon` 186
`Ray::d` 67
`Ray::maxt` 67
`Ray::o` 67

For example, when traversing a spatial data structure, if the ray doesn't pass through the node that holds the geometry, then at some parent of that node the traversal code will make an incorrect decision and decide to not recurse down to the subtree that has the node with the intersection. Finding the node where the intersection occurs and from that

the path from the root of the tree to that node makes it possible to isolate where this error happens. Alternatively, if the ray passes through that node but the primitive isn't present in it, then the construction code needs to be examined to figure out why the primitive wasn't included in the node.

For accelerators based on primitive subdivision, the task is similar. The primitive that should have been hit will be present in one or more nodes of the data struture; the question again is "Why didn't the ray visit those nodes, or if it did, then why wasn't an intersection found?"

Drilling down to the source of these bugs can often be done more easily by instrumenting the code with `printf()` calls that show what the system is doing (as it traverses the data structure or builds it, respectively). We have found that after narrowing down to a scene and ray that exhibits a bug, that a mixture of both printing detailed information about the code's execution and stepping through execution in a debugger is effective. One advantage of printing out a trace of the execution of the code is that it can be easier to work forward and backward through the trace file to find the source of error.

4.6.3 AGGREGATE PERFORMANCE BUGS

Beyond correctness bugs, errors that cause performance problems can be nefarious and difficult to find—with these bugs, although the system still computes the correct result, it just does so very inefficiently. An example of this type of bug was present in the first version of the `pbrt` system: due to a subtle bug in code that computed specular refraction directions, rays with floating-point "not a number" values for their direction components would very rarely be generated when rendering the ecosystem scene in Figure 4.1. When given a ray with NaN direction components, the kd-tree accelerator traversal code will visit *every* node in the entire tree. For the ecosystem scene, this meant that these rays would be tested for intersection with all 19 million triangles in the scene. This is obviously extremely wasteful, but it only happened for a handful of rays in the scene, so the overall rendering time wasn't sufficiently bad to be obviously wrong. When we found and fixed this bug, performance rendering that scene increased by a factor of three.

The best approach we have found to finding these sorts of bugs is to liberally gather statistics about the code's execution and to use visualization tools to understand its behavior. For example, simple code to track the maximum number of ray–primitive intersection tests for all the rays traced during rendering would have made it obvious that, for at least one ray, over 19 million intersection tests were being performed. From this insight, working backward to figure out which ray was the culprit and then why this was the case would have been relatively straightforward.

FURTHER READING

After the introduction of the ray-tracing algorithm, an enormous amount of research was done to try to find effective ways to speed it up, primarily by developing improved ray-tracing acceleration structures. Arvo and Kirk's chapter in *An Introduction to Ray Tracing* (Glassner 1989a) summarizes the state of the art as of 1989 and still provides an

excellent taxonomy for categorizing different approaches to ray intersection acceleration. *Ray Tracing News* (*www.acm.org/tog/resources/RTNews/*) is a very good resource for general ray-tracing information and has particularly useful discussions about intersection acceleration approaches, implementation issues, and tricks of the trade.

Kirk and Arvo (1988) introduced the unifying principle of *meta-hierarchies*. They showed that by implementing acceleration data structures to conform to the same interface as is used for primitives in the scene, it's easy to mix and match multiple intersection acceleration schemes. `pbrt` follows this model since the `Aggregate` inherits from the `Primitive` base class.

In the interests of making it easier to compare the performance of different ray intersection algorithms, there have been some efforts to create standard databases of scenes to test various ray intersection algorithms, notably Haines's "standard procedural database" (SPD) (Haines 1987) and Lext et al.'s BART scenes, which include animation (Lext, Assarsson, and Möller 2001). A few of the SPD scenes are available in the `pbrt` file format in the `pbrt` distribution.

Grids

Fujimoto, Tanaka, and Iwata (1986) were the first to introduce uniform voxel grids for ray tracing, similar to the approach implemented in this chapter. Snyder and Barr (1987) described a number of key improvements to this approach and showed their use for rendering extremely complex scenes. Hierarchical grids were first described by Jevans and Wyvill (1989). More complex techniques for hierarchical grids were developed by Cazals, Drettakis, and Puech (1995) and Klimaszewski and Sederberg (1997). The grid traversal method used in this chapter is essentially the one described by Cleary and Wyvill (1988).

Choosing an optimal grid resolution has received attention from a number of researchers. A recent paper in this area is by Ize et al. (2007), who provided a solid foundation for selecting an optimal grid resolution and for deciding when to refine into subgrids, when hierarchical grids are being used. They derived theoretical results using a number of simplifying assumptions and then showed the applicability of the results to rendering real-world scenes. They also included a good selection of pointers to previous work in this area.

Lagae and Dutré (2008a) described an innovative representation for uniform grids that has the desirable properties that not only does each primitive have a single index into a voxel, but each voxel has only a single primitive index. They show that this representation has very low memory usage and is still quite efficient.

Hunt and Mark (2008) showed that building grids in perspective space, where the center of projection is the camera or a light source, can make tracing rays from the camera or light substantially more efficient. Although this approach requires multiple acceleration structures, the performance benefits from multiple specialized structures for different classes of rays can be substantial. Their approach is also notable in that it is in some ways a middle-ground between rasterization and ray tracing.

Aggregate 192
Primitive 185

Bounding Volume Hierarchies

Clark (1976) first suggested using bounding volumes to cull collections of objects for standard visible-surface determination algorithms. Building on this work, Rubin and

Whitted (1980) developed the first hierarchical data structures for scene representation for fast ray tracing, although their method depended on the user to define the hierarchy. Kay and Kajiya (1986), implemented one of the first practical object subdivision approaches based on bounding objects with collections of slabs.

Goldsmith and Salmon (1987) described an algorithm for automatically computing bounding volume hierarchies and applied techniques for estimating the probability of a ray intersecting a bounding volume based on the volume's surface area. Most current methods for building BVHs are based on top-down construction of the tree, first creating the root node and then partitioning the primitives into children and continuing recursively. An alternative approach was demonstrated by Walter et al. (2008), who showed that bottom-up construction, where the leaves are created first and then agglomerated into parent nodes, is viable and can build somewhat better trees than top-down approaches. Kensler (2008) presented algorithms that make local adjustments to the BVH tree after it has been built to improve its quality.

The `BVHAccel` implementation in this chapter is based on the construction algorithm described by Wald (2007) and Gunther et al. (2007). The bounding box test is the one introduced by Williams et al. (2005). An even more efficient bounding box test that does additional precomputation in exchange for higher performance when the same ray is tested for intersection against many bounding boxes was developed by Eisemann et al. (2007); we leave implementing their method for an exercise.

The BVH traversal algorithm used in `pbrt` was concurrently developed by a number of researchers; see the notes by Boulos and Haines (2006) for more details and background. Another option for tree traversal is that of Kay and Kajiya (1986); they maintained a heap of nodes ordered by ray distance.

One shortcoming of BVHs is that even a small number of relatively large primitives that have overlapping bounding boxes can substantially reduce the efficiency of the BVH: many of the nodes of the tree will be overlapping, solely due to the overlapping bounding boxes of geometry down at the leaves. Ernst and Greiner (2007) proposed "split clipping" as a solution to this problem; the restriction that each primitive only appears once in the tree is lifted, and the bounding boxes of large input primitives are subdivided into a set of tighter sub-bounds which are then used for tree construction. This happens only during the tree construction and doesn't affect the tree representation or the rendering algorithm. Dammertz and Keller (2008) observed that the problematic primitives are the ones with a large amount of empty space in their bounding box relative to their surface area, so they subdivided the most egregious triangles and reported substantial performance improvements. Stich et al. (2009) developed an approach that splits primitives during BVH construction, rather than as a preprocess, making it possible to only split primitives when a SAH cost reduction was found. See also Popov et al.'s recent paper on a theoretically optimum BVH partitioning algorithm and its relationship to previous approaches (Popov et al. 2009).

The memory requirements for BVHs can be substantial. In our implementation, each node is 32 bytes. With up to 2 BVH tree nodes needed per primitive in the scene, the total overhead may be as high as 64 bytes per primitive. Cline et al. (2006) suggested a more compact representation for BVH nodes, at some expense of efficiency. First, they quan-

BVHAccel 209

tized the bounding box stored in each node using 8 or 16 bytes to offset with respect to the bounding box of the entire tree. Second, they used *implicit indexing*, where the node i's children are at positions $2i$ and $2i + 1$ in the node array (assuming a $2\times$ branching factor). They showed substantial memory savings, with moderate performance impact. See also Mahovsky's Ph.D. thesis (2005) for other approaches to reducing BVH memory usage.

Yoon and Manocha (2006) described algorithms for cache-efficient layout of BVHs and kd-trees and demonstrated performance improvements from doing so. See also Ericson's book (2004) for extensive discussion of this topic.

kd-trees

Glassner (1984) introduced the use of octrees for ray intersection acceleration; this approach was more robust for scenes with nonuniform distributions of geometry than grids. Use of the kd-tree was first described by Kaplan (1985). Kaplan's tree construction algorithm always split nodes down their middle; MacDonald and Booth (1990) introduced the surface area heuristic approach, estimating ray–node traversal probabilities using relative surface areas. Naylor (1993) has also written on general issues of constructing good kd-trees. Havran and Bittner (2002) revisited many of these issues and introduced useful improvements. Adding a bonus factor to the surface area heuristic for tree nodes that are completely empty, as is done in our implementation, was suggested by Hurley et al. (2002).

Jansen (1986) first developed the efficient ray traversal algorithm for kd-trees. Arvo also investigated this problem and discussed it in a note in *Ray Tracing News* (Arvo 1988). Sung and Shirley (1992) described a ray traversal algorithm's implementation for a BSP-tree accelerator; our `KdTreeAccel` traversal code is loosely based on theirs.

The asymptotic complexity of the kd-tree construction algorithm in `pbrt` is $O(n \log^2 n)$. Wald and Havran (2006) showed that it's possible to build kd-trees in $(n \log n)$ time with some additional implementation complexity; they reported a 2 to $3\times$ speedup in construction time for typical scenes.

The best kd-trees for ray tracing are built using "perfect splits," where the primitive being inserted into the tree is clipped to the bounds of the current node at each step. This eliminates the issue that, for example, an object's bounding box may intersect a node's bounding box and thus be stored in it, even though the object itself doesn't intersect the node's bounding box. This approach was introduced by Havran and Bittner (2002) and discussed further by Hurley et al. (2002) and Wald and Havran (2006). See also Soupikov et al. (2008).

kd-tree construction tends to be much slower than BVH construction (especially if "perfect splits" are used), so parallel construction algorithms are of particular interest. Recent work in this area includes that of Shevtsov et al. (2007b), who presented an efficient parallel kd-tree construction algorithm with good scalability to multiple processors.

The Surface Area Heuristic

KdTreeAccel 228

A number of researchers have investigated improvements to the surface area heuristic since its introduction to ray tracing by MacDonald and Booth (1990). Fabianowski et al. (2009) derived a version that replaces the assumption that rays are uniformly distributed

throughout space with the assumption that ray origins are uniformly distributed inside the scene's bounding box. Hunt and Mark (2008b) built specialized acceleration structures for the camera and each light, using a perspective projection to warp space as seen from the camera or light. This warping led to a new SAH that better accounts for the fact that the rays aren't in fact uniformly distributed but that a large number of them originate from a single point or a set of nearby points (for depth of field and area light sources, respectively). Hunt (2008) showed how the SAH should be modified when the "mailboxing" optimization is being used.

Evaluating the SAH can be costly, particularly when many different splits or primitive partitions are being considered. One solution to this problem is to only compute it at a subset of the candidate points—for example, along the lines of the bucketing approach used in the `BVHAccel` in `pbrt`. Hurley et al. (2002) suggested this approach for building kd-trees, and Popov et al. (2006) applied it to kd-trees. Shevtsov et al. (2007) introduced the improvement of binning the full extents of triangles, not just their centroids.

Hunt et al. (2006) noted that if you only have to evaluate the SAH at one point, for example, you don't need to sort the primitives, but only need to do a linear scan over them to compute primitve counts and bounding boxes on each point. They showed that approximating the SAH with a piecewise quadratic based on evaluating it at a number of individual positions and using that to choose a good split leads to effective trees. A similar approximation was used by Popov et al. (2006).

Other Topics in Acceleration Structures

Weghorst, Hooper, and Greenberg (1984) discussed the trade-offs of using various shapes for bounding volumes and suggested projecting objects to the screen and using a z-buffer rendering to accelerate finding intersections for camera rays.

A number of researchers have investigated the applicability of general BSP trees, where the splitting planes aren't necessarily axis aligned, as they are with kd-trees. Kammaje and Mora (2007) built BSP trees using a preselected set of candidate splitting planes, and Budge et al. (2008) developed a number of improvements to their approach, though only approached kd-tree performance in practice due to a slower construction stage and slower traversal than kd-trees. Ize et al. (2008) showed a BSP implementation that renders scenes faster than modern kd-trees, but at the cost of extremely long construction times.

There are many techniques for traversing a collection of rays through the acceleration structure together, rather than just one at a time. This approach ("packet tracing") is an important component of high-performance ray tracing; it's discussed in more depth in Section 18.2. Another major area of recent research has been acceleration structures that can be incrementally updated over frames of an animation, rather than requiring reconstruction from scratch. See, for example, Wald et al. (2007) for recent work in this area.

An innovative approach was suggested by Arvo and Kirk (1987), who introduced a five-dimensional data structure that subdivided based on both 3D spatial and 2D ray directions. Another interesting approach for scenes described with triangle meshes was developed by Lagae and Dutré (2008b): they computed a constrained tetrahedralization, where all triangle faces of the model are represented in the tetrahedralization. Rays are then stepped through tetrahedra until they intersect a triangle from the scene descrip-

BVHAccel 209

tion. This approach is still a few times slower than the state-of-the-art in kd-trees and BVHs but is an interesting new way to think about the problem.

There is an interesting middle-ground between kd-trees and BVHs, where the tree node holds a splitting plane for each child, rather than just a single splitting plane. For example, this refinement makes it possible to do object subdivision in a kd-tree-like acceleration structure, putting each primitive in just one subtree and allowing the subtrees to overlap, while still preserving many of the benefits of efficient kd-tree traversal. Ooi et al. (1987) first introduced this refinement to kd-trees for storing spatial data, naming it the "spatial kd-tree" (skd-tree). Skd-trees have recently been applied to ray tracing by a number of researchers, including Zachmann (2002), Woop et al. (2006), Wächter and Keller (2006), Havran et al. (2006), and Zuniga and Uhlmann (2006).

When spatial subdivision is used, primitives may overlap multiple nodes of the structure and a ray may be tested for intersection with the same primitive multiple times as it passes through the structure. Arnaldi, Priol, and Bouatouch (1987) and Amanatides and Woo (1987) developed the "mailboxing" technique to address this issue: each ray is given a unique integer identifier and each primitive records the id of the last ray that was tested against it. If the ids match, then the intersection test is unnecessary and can be skipped.

EXERCISES

4.1 What kind of scenes are worst-case scenarios for the three acceleration structures in `pbrt`? (Consider specific geometric configurations that the approaches will respectively be unable to handle well.) Construct scenes with these characteristics, and measure the performance of `pbrt` as you add more primitives. How does the worst case for one behave when rendered with the others?

4.2 Read the paper by Ize et al. (2007) and apply their methods for selecting grid resolution to the `GridAccel` in this chapter. Measure the trade-offs related to time spent building the grid, memory used to represent the grid, and time needed to find ray–object intersections for different grid resolutions.

4.3 Generalize the grid implementation in this chapter to be hierarchical: refine voxels that have an excessive number of primitives overlapping them to instead hold a finer subgrid to store its geometry. (See, for example, Jevans and Wyvill (1989) for one approach to this problem and Ize et al. (2007) for effective methods for deciding when refinement is worthwhile.)

4.4 Implement the compact grid representation introduced by Lagae and Dutré (2008a). How does the performance of your implementation compare to the `GridAccel` in `pbrt`? (Measure construction time, memory use, and time to find ray intersections.)

4.5 Implement smarter overlap tests for building accelerators. Using objects' bounding boxes to determine which grid cells and which sides of a kd-tree split they overlap can hurt performance by causing unnecessary intersection tests. (Recall Figure 4.5.) Add a `bool Shape::Overlaps(const BBox &) const` method

GridAccel 196

to the shape interface that takes a world space bounding box and determines if the shape truly overlaps the given bound.

A default implementation could get the world bound from the shape and use that for the test, and specialized versions could be written for frequently used shapes. Implement this method for `Spheres` and `Triangles` and modify the accelerators to call it. You may find it helpful to read Akenine-Möller's paper on fast triangle-box overlap testing (Akenine-Möller 2001). Measure the change in `pbrt`'s overall performance due to this change, separately accounting for increased time spent building the acceleration structure and reduction in ray–object intersection time due to fewer intersections. For a variety of scenes, determine how many fewer intersection tests are performed thanks to this improvement.

4.6 Implement "split clipping" in `pbrt`'s BVH implementation. Read the papers by both Ernst and Greiner (2007) and Dammertz and Keller (2008) and implement one of their approaches to subdivide primitives with large bounding boxes relative to their surface area into multiple subprimitives for tree construction. (Doing so will probably require modification to the `Shape` interface; you will probably want to design a new interface that allows some shapes to indicate that they are unable to subdivide themselves, so that you only need to implement this method for triangles, for example.) Measure the improvement for rendering actual scenes; a compelling way to gather this data is to do the experiment that Dammertz and Keller did, where a scene is rotated around an axis over progressive frames of an animation. Typically, many triangles that are originally axis aligned will have very loose bounding boxes as they rotate more, leading to a substantial performance degradation if split clipping isn't used.

4.7 Fix either the `BVHAccel` or the `KdTreeAccel` so that it doesn't always immediately refine all primitives before building the tree but instead builds subtrees on demand. Care must be taken when updating the data structures in the presence of multi-threading so that other threads don't see the tree in an inconsistent state as it is being updated. One option is to use a reader-writer mutex, as the `GridAccel` does, though the cost of acquiring the mutex for each ray is significant. More efficient is to use a lock-free approach for updating the data structure, as described in Section A.9.2.

4.8 On systems with 64-bit pointers, the `KdAccelNode` structure will actually be 12 bytes large, thanks to an 8-byte pointer for the `KdAccelNode::primitives` array. Modify the implementation to fix this problem. One approach would be to allocate all of the memory for all of the `primitives` arrays contiguously—for example, with a `vector` stored in the `KdTreeAccel`. Then, nodes of the tree that had multiple primitives would store an offset into this vector where their primitive numbers started, rather than a pointer. How much does this change affect performance in practice for reasonably complex scenes that use a kd-tree accelerator?

BVHAccel 209
GridAccel 196
KdAccelNode 229
KdAccelNode::primitives 229
KdTreeAccel 228
Shape 108

4.9 Investigate alternative SAH cost functions for building BVHs or kd-trees. How much can a poor cost function hurt its performance? How much improvement

can be had compared to the current one? (See the discussion in the "Further Reading" section for ideas about how the SAH may be improved.)

● 4.10 Construction time for the `BVHAccel` and particularly the `KdTreeAccel` can be a meaningful portion of overall rendering time, yet the implementations in this chapter do not parallelize building the acceleration structures. Investigate techniques for parallel construction of accelerators such as described by Wald (2007) and Shevtsov et al. (2007) and implement one of them in `pbrt`. How much of a speedup do you achieve in accelerator construction? How does the speedup scale with additional processors? Measure how much of a speedup your changes translate to for overall rendering. For what types of scenes does your implementation have the greatest impact?

● 4.11 The idea of using spatial data structures for ray intersection acceleration can be generalized to include spatial data structures that themselves hold other spatial data structures, rather than just primitives. Not only could we have a grid that has subgrids inside the grid cells that have many primitives in them (thus partially solving the adaptive refinement problem), but we could also have the scene organized into a hierarchical bounding volume where the leaf nodes are grids that hold smaller collections of spatially nearby primitives. Such hybrid techniques can bring the best of a variety of spatial data structure-based ray intersection acceleration methods. In `pbrt`, because both geometric primitives and intersection accelerators inherit from the `Primitive` base class and thus provide the same interface, it's easy to mix and match in this way.

Modify `pbrt` to build hybrid acceleration structures—for example, using a BVH to coarsely sort the scene geometry and then uniform grids at the leaves of the tree to manage dense, spatially local collections of geometry. Measure the running time and memory use for rendering schemes with this method compared to the current accelerators.

● 4.12 Eisemann et al. (2007) described an even more efficient ray–box intersection test than is used in the `BVHAccel`. It does more computation at the start for each ray, but makes up for this work with fewer computations to do tests for individual bounding boxes. Implement their method in `pbrt` and measure the change in rendering time for a variety of scenes. Are there simple scenes where the additional upfront work doesn't pay off? How does the improvement for highly complex scenes compare to the improvement for simpler scenes?

● 4.13 It is often possible to introduce some approximation into the computation of shadows from very complex geometry (consider, for example, the branches and leaves of a tree casting a shadow). Lacewell et al. (2008) suggested augmenting the acceleration structure with a prefiltered directionally varying representation of occlusion for regions of space. As shadow rays pass through these regions, an approximate visibility probability can be returned rather than a binary result, and the cost of tree traversal and object intersection tests is reduced. Implement this approach in `pbrt` and measure its performance.

AggregateTest 247
BVHAccel 209
KdTreeAccel 228
Primitive 185

● 4.14 The automated testing code in `AggregateTest` can be an effective way to find test cases that show a scene and a particular ray that exhibit a bug. For very complex

scenes, however, debugging these failures can be tedious; if a simpler scene could be found that exhibited the same bug, programmer time can be saved. Investigate automated techniques for finding minimal reproduction cases for bugs. For example, Zeller and Hildebrandt (2002) describe an algorithm that automatically tries to maximally simplify failure-inducing input; their method could be applied to try to remove primitives from the scene description and find a simpler scene that still had an error.

4.15 In addition to using existing scenes to test accelerators, randomly generated scenes can also be effective at finding bugs. In general, the search space for accelerator bugs is enormous, though the number of available processing cycles is also large. (Running random tests on accelerators in the background for weeks or months can be worthwhile in that bugs found in this manner are easier to track down than running into them when rendering a scene and seeing an unexpected image artifact.) Implement code that randomly generates scenes for testing. Effective approaches include building scenes from existing models, randomly transforming them to place them, perturbing existing models (for example, changing some vertex positions of a triangle mesh), or generating completely random scenes (for example, a random number of triangles, each with random vertex positions). Can you find bugs in `pbrt`'s accelerators with this technique?

4.16 Modify the system to record all rays that were traced while rendering a scene to a file and then use rays gathered in this way to test the aggregates. Modify the `AggregateTest` to read files of these rays and use them for its tests. The easiest way to collect rays is probably to modify the `Scene::Intersect()` and `IntersectP()` methods to save them. Be careful to use either the `%a` formatting string described in Section 4.6.1 for accurately writing floating-point values to a file, or write the raw bits of the `float`, for example, using `fwrite()`. Also be aware of the implications of multi-threading: if multiple threads write to the same file, their output can be interleaved. Either use a mutex to protect access to the file, or be sure to store all of the values for a ray with a single `fprintf()` or `fwrite()` call, rather than using multiple calls to write all of the components. (Implementations of the C standard library do guarantee atomicity from single calls to those functions, using a mutex internally.) Try to fix any bugs you find in aggregates.

AggregateTest 247
Scene::Intersect() 23

This page intentionally left blank



CHAPTER FIVE

05 COLOR AND RADIOMETRY

In order to precisely describe how light is represented and sampled to compute images, we must first establish some background in *radiometry*—the study of the propagation of electromagnetic radiation in an environment. Of particular interest in rendering are the wavelengths (λ) of electromagnetic radiation between approximately 370 nm and 730 nm, which account for light visible to humans. The lower wavelengths ($\lambda \approx 400$ nm) are the bluish colors, the middle wavelengths ($\lambda \approx 550$ nm) are the greens, and the upper wavelengths ($\lambda \approx 650$ nm) are the reds.

In this chapter, we will introduce four key quantities that describe electromagnetic radiation: flux, intensity, irradiance, and radiance. These radiometric quantities are each described by their *spectral power distribution* (SPD)—a distribution function of wavelength that describes the amount of light at each wavelength. The `Spectrum` class, which is defined in Section 5.1, is used to represent SPDs in `pbrt`.

5.1 SPECTRAL REPRESENTATION

The SPDs of real-world objects can be quite complicated; Figure 5.1 shows graphs of the spectral distribution of emission from a fluorescent light and the spectral distribution of the reflectance of lemon skin. A renderer doing computations with SPDs needs a compact, efficient, and accurate way to represent functions like these. In practice, some trade-off needs to be made between these qualities.

A general framework for investigating these issues can be developed based on the problem of finding good *basis functions* to represent SPDs. The idea behind basis functions is to map the infinite-dimensional space of possible SPD functions to a low-dimensional space of coefficients $c_i \in \mathbb{R}$. For example, a trivial basis function is the constant function $B(\lambda) = 1$. An arbitrary SPD would be represented in this basis by a single coefficient c equal to its average value, so that its approximation would be $cB(\lambda) = c$. This is obviously

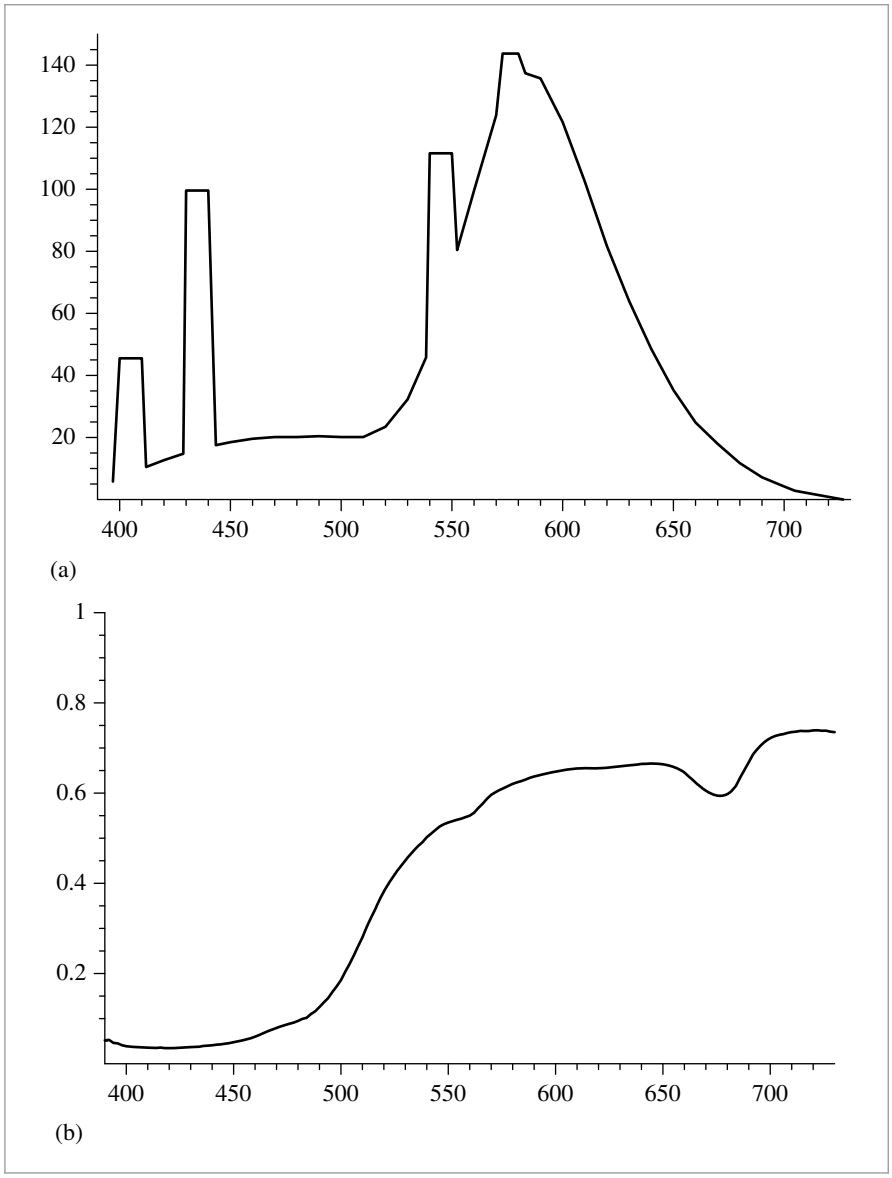


Figure 5.1: (a) Spectral power distributions of a fluorescent light and (b) the reflectance of lemon skin. Wavelengths around 400 nm are bluish colors, greens and yellows are in the middle range of wavelengths, and reds have wavelengths around 700 nm. The fluorescent light's SPD is even spikier than shown here, where the SPDs have been binned into 10 nm ranges; it actually emits much of its illumination at single discrete frequencies.

a poor approximation, since most SPDs are much more complex than this single-basis function is capable of representing accurately.

Many different basis functions have been investigated for spectral representation in computer graphics; the "Further Reading" section cites a number of papers and further resources on this topic. Different sets of basis functions can offset substantially different trade-offs in the complexity of the key operations like converting an arbitrary SPD into a set of coefficients (projecting it into the basis), computing the coefficients for the SPD given by the product of two SPDs expressed in the basis, and so on. In this chapter, we'll introduce two representations that can be used for spectra in `pbrt`: `RGBSpectrum`, which follows the typical computer graphics practice of representing SPDs with coefficients representing red, green, and blue colors, and `SampledSpectrum`, which represents the spectrum as a set of point samples over a range of wavelengths.

5.1.1 THE `Spectrum` TYPE

Throughout `pbrt`, we have been careful to implement all computations involving SPDs in terms of the `Spectrum` type, expecting a particular set of built-in operators (addition, multiplication, etc.). The `Spectrum` type hides the details of the particular spectral representation used, so that changing this detail of the system only requires changing the `Spectrum` implementation; other code can remain unchanged. The implementations of the `Spectrum` type are in the files `core/spectrum.h` and `core/spectrum.cpp`.

The selection of which spectrum representation is used for the `Spectrum` type in `pbrt` is done with a `typedef` in the file `core/pbrt.h`. By default, `pbrt` uses the more efficient but less accurate RGB representation.

⟨*Global Forward Declarations*⟩ ≡
```
typedef RGBSpectrum Spectrum;
// typedef SampledSpectrum Spectrum;
```

We have not written the system such that the selection of which `Spectrum` implementation to use could be resolved at run time; if the implementation is changed, the entire system must be recompiled. One advantage to this design is that many of the various `Spectrum` methods can be implemented as short functions that can be inlined by the compiler, rather than being left as stand-alone functions that have to be invoked through the slow virtual method call mechanism. Inlining frequently used short functions like these can give a substantial improvement in performance. A second advantage is that structures in the system that hold instances of the `Spectrum` type can hold them directly, rather than needing to allocate them dynamically based on the spectral representation chosen at run time.

5.1.2 `CoefficientSpectrum` IMPLEMENTATION

`CoefficientSpectrum` 264
`RGBSpectrum` 279
`SampledSpectrum` 266
`Spectrum` 263

Both of the representations implemented in this chapter are based on storing a fixed number of samples of the SPD. Therefore, we'll start by defining the `CoefficientSpectrum` template class, which represents a spectrum as by a particular number of samples given by the `nSamples` template parameter. Both `RGBSpectrum` and `SampledSpectrum` are built on top of it.

⟨*Spectrum Declarations*⟩ ≡
```
template <int nSamples> class CoefficientSpectrum {
public:
    ⟨CoefficientSpectrum Public Methods 264⟩
protected:
    ⟨CoefficientSpectrum Protected Data 264⟩
};
```

One `CoefficientSpectrum` constructor is provided; it initializes a spectrum with a constant value across all wavelengths.

⟨*CoefficientSpectrum Public Methods*⟩ ≡ 264
```
CoefficientSpectrum(float v = 0.f) {
    for (int i = 0; i < nSamples; ++i)
        c[i] = v;
}
```

⟨*CoefficientSpectrum Protected Data*⟩ ≡ 264
```
float c[nSamples];
```

A variety of arithmetic operations on `Spectrum` objects are needed; the implementations in `CoefficientSpectrum` are all straightforward. First are operations to add pairs of spectral distributions. For the sampled representation, it's easy to show that each sample value for the sum of two SPDs is equal to the sum of the corresponding sample values.

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
CoefficientSpectrum &operator+=(const CoefficientSpectrum &s2) {
    for (int i = 0; i < nSamples; ++i)
        c[i] += s2.c[i];
    return *this;
}
```

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
CoefficientSpectrum operator+(const CoefficientSpectrum &s2) const {
    CoefficientSpectrum ret = *this;
    for (int i = 0; i < nSamples; ++i)
        ret.c[i] += s2.c[i];
    return ret;
}
```

Similarly, subtraction, multiplication, division, and unary negation are defined component-wise. These methods are very similar to the ones already shown, so we won't include them here. `pbrt` also provides equality and inequality tests, also not included here.

It is often useful to know if a spectrum represents an SPD with value zero everywhere. If, for example, a surface has zero reflectance, the light transport routines can avoid the computational cost of casting reflection rays that have contributions that would eventually be multiplied by zeros and thus do not need to be traced.

CoefficientSpectrum 264
CoefficientSpectrum::c 264
Spectrum 263

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
bool IsBlack() const {
    for (int i = 0; i < nSamples; ++i)
        if (c[i] != 0.) return false;
    return true;
}
```

The `Spectrum` implementation (and thus the `CoefficientSpectrum` implementation) must also provide implementations of a number of slightly more esoteric methods, including ones that take the square root of a spectrum or raise the components of a `Spectrum` to a given power. These are needed for some of the computations performed by the `Fresnel` classes in Chapter 8. The implementation of `Sqrt()` takes the square root of each component to give the square root of the SPD. The implementations of `Pow()` and `Exp()` are analoguous and won't be included here.

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
friend CoefficientSpectrum Sqrt(const CoefficientSpectrum &s) {
    CoefficientSpectrum ret;
    for (int i = 0; i < nSamples; ++i)
        ret.c[i] = sqrtf(s.c[i]);
    return ret;
}
```

It's frequently useful to be able to linearly interpolate between two SPDs with a parameter t.

⟨*Spectrum Inline Functions*⟩ ≡
```
inline Spectrum Lerp(float t, const Spectrum &s1, const Spectrum &s2) {
    return (1.f - t) * s1 + t * s2;
}
```

Some portions of the image processing pipeline will want to clamp a spectrum to ensure that the function it represents is within some allowable range.

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
CoefficientSpectrum Clamp(float low = 0, float high = INFINITY) const {
    CoefficientSpectrum ret;
    for (int i = 0; i < nSamples; ++i)
        ret.c[i] = ::Clamp(c[i], low, high);
    return ret;
}
```

Finally, we provide a debugging routine to check if any of the sample values of the SPD is the not-a-number floating-point value (NaN). This situation can happen due to an accidental division by zero; `Assert()`s throughout the system use this method to catch this case close to where it happens.

Assert() 1005
Clamp() 1000
CoefficientSpectrum 264
CoefficientSpectrum::c 264
Fresnel 436
INFINITY 1002
Spectrum 263

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
bool HasNaNs() const {
    for (int i = 0; i < nSamples; ++i)
        if (isnan(c[i])) return true;
    return false;
}
```

5.2 THE SampledSpectrum CLASS

`SampledSpectrum` uses the `CoefficientSpectrum` infrastructure to represent a SPD with uniformly spaced samples between a starting and an ending wavelength. The wavelength range covers from 400 nm to 700 nm—the range of the visual spectrum where the human visual system is most sensitive. The number of samples, 30, is generally more than enough to accurately represent complex SPDs for rendering. Thus, the first sample represents the wavelength range [400, 410), the second represents [410, 420), and so forth. These values can easily be changed here as needed.

⟨*Spectrum Utility Declarations*⟩ ≡
```
static const int sampledLambdaStart = 400;
static const int sampledLambdaEnd = 700;
static const int nSpectralSamples = 30;
```

⟨*Spectrum Declarations*⟩ +≡
```
class SampledSpectrum : public CoefficientSpectrum<nSpectralSamples> {
public:
    ⟨SampledSpectrum Public Methods 266⟩
private:
    ⟨SampledSpectrum Private Data 271⟩
};
```

By inheriting from the `CoefficientSpectrum` class, `SampledSpectrum` automatically has all of the basic spectrum arithmetic operations defined earlier. The main methods left to define for it are those that initialize it from spectral data and convert the SPD it represents to other spectral representations (such as RGB). The constructor for initializing it with a constant SPD is straightforward.

⟨*SampledSpectrum Public Methods*⟩ ≡ 266
```
SampledSpectrum(float v = 0.f) {
    for (int i = 0; i < nSpectralSamples; ++i) c[i] = v;
}
```

We will often be provided spectral data as a set of (λ_i, v_i) samples, where the ith sample has some value v_i at wavelength λ_i. In general, the samples may have irregular spacing and there may be more or fewer of them than the `SampledSpectrum` stores. (See the directory `scenes/spds` in the `pbrt` distribution for a variety of SPDs for use with `pbrt`, many of them with irregular sample spacing. The file `docs/fileformat.pdf` describes how to use measured spectral data in `pbrt`'s scene description files.)

CoefficientSpectrum 264
CoefficientSpectrum::c 264
nSpectralSamples 266
SampledSpectrum 266

The FromSampled() method takes arrays of SPD sample values v at given wavelengths lambda and uses them to define a piecewise linear function to represent the SPD. For each SPD sample in the SampledSpectrum, it uses the AverageSpectrumSamples() utility function, defined below, to compute the average of the piecewise linear function over the range of wavelengths that each SPD sample is responsible for.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
static SampledSpectrum FromSampled(const float *lambda,
                                   const float *v, int n) {
    ⟨Sort samples if unordered, use sorted for returned spectrum 267⟩
    SampledSpectrum r;
    for (int i = 0; i < nSpectralSamples; ++i) {
        ⟨Compute average value of given SPD over ith sample's range 267⟩
    }
    return r;
}
```

The AverageSpectrumSamples() function requires that the (λ_i, v_i) values be sorted by wavelength. The SpectrumSamplesSorted() function checks that they are; if not, SortSpectrumSamples() sorts them. Note that we allocate new storage for the sorted samples and do not modify the values passed in by the caller; in general, doing so would likely be unexpected behavior for a user of this function (who shouldn't need to worry about these requirements of its specific implementation). We won't include the implementations of either of these two functions here, as they are straightforward.

⟨*Sort samples if unordered, use sorted for returned spectrum*⟩ ≡ 267, 280
```
if (!SpectrumSamplesSorted(lambda, v, n)) {
    vector<float> slambda(&lambda[0], &lambda[n]);
    vector<float> sv(&v[0], &v[n]);
    SortSpectrumSamples(&slambda[0], &sv[0], n);
    return FromSampled(&slambda[0], &sv[0], n);
}
```

In order to compute the value for the ith spectral sample, we compute the range of wavelengths that it's responsible for—lambda0 to lambda1—and use the AverageSpectrumSamples() function to compute the average value of the given piecewise linear SPD over that range. This is a one-dimensional instance of sampling and reconstruction, a topic that will be discussed in more detail in Chapter 7.

⟨*Compute average value of given SPD over ith sample's range*⟩ ≡ 267
```
float lambda0 = Lerp(float(i) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
float lambda1 = Lerp(float(i+1) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
r.c[i] = AverageSpectrumSamples(lambda, v, n, lambda0, lambda1);
```

Figure 5.2 shows the basic approach taken by AverageSpectrumSamples(): it iterates over each of the linear segments between samples that are partially or fully within the range of wavelengths, lambdaStart to lambdaEnd. For each such segment, it computes the average value over its range, scales the average by the wavelength range the segment covers, and

AverageSpectrumSamples() 268
CoefficientSpectrum::c 264
Lerp() 1000
nSpectralSamples 266
sampledLambdaEnd 266
sampledLambdaStart 266
SampledSpectrum 266
SampledSpectrum::
FromSampled() 267
SortSpectrumSamples() 267
SpectrumSamplesSorted() 267

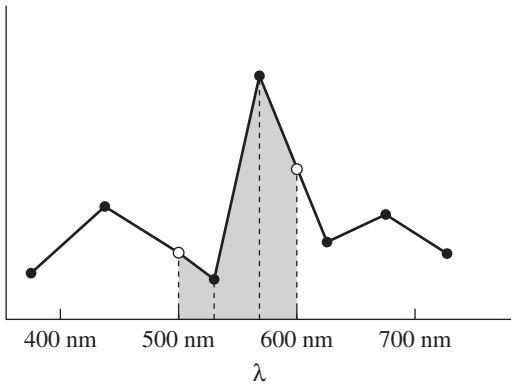


Figure 5.2: When resampling an irregularly defined SPD, we need to compute the average value of the piecewise linear function defined by the SPD samples. Here, we want to average the value from 500 nm to 600 nm—the shaded region under the plot. The `FromSampled()` function does this by computing the areas of each of the regions denoted by dashed lines in this figure.

accumulates a sum of these values. The final average value is this sum divided by the total wavelength range.

⟨*Spectrum Method Definitions*⟩ ≡
```
float AverageSpectrumSamples(const float *lambda, const float *vals,
        int n, float lambdaStart, float lambdaEnd) {
    ⟨Handle cases with out-of-bounds range or single sample only 268⟩
    float sum = 0.f;
    ⟨Add contributions of constant segments before/after samples 269⟩
    ⟨Advance to first relevant wavelength segment 269⟩
    ⟨Loop over wavelength sample segments and add contributions 269⟩
    return sum / (lambdaEnd - lambdaStart);
}
```

The function starts by checking for and handling the edge cases where the range of wavelengths to average over is outside the range of provided wavelengths or the case where there is only a single sample, in which case the average value is trivially computed. We assume that the SPD has a constant value (the values at the two respective end points) outside of the provided sample range; if this isn't a reasonable assumption for a particular set of data, the data provided should have explicit values of (for example) zero at the end points.

⟨*Handle cases with out-of-bounds range or single sample only*⟩ ≡ 268
```
if (lambdaEnd   <= lambda[0])   return vals[0];
if (lambdaStart >= lambda[n-1]) return vals[n-1];
if (n == 1) return vals[0];
```

Having handled these cases, the next step is to check to see if part of the range to average over goes beyond the first and/or last sample value. If so, we accumulate the contribution of the constant segment(s), scaled by the out-of-bounds wavelength range.

⟨*Add contributions of constant segments before/after samples*⟩ ≡ 268
```
if (lambdaStart < lambda[0])
    sum += vals[0] * (lambda[0] - lambdaStart);
if (lambdaEnd > lambda[n-1])
    sum += vals[n-1] * (lambdaEnd - lambda[n-1]);
```

And now we advance to the first index `i` where the starting wavelength of the interpolation range overlaps the segment from λ_i to λ_{i+1}. A more efficient implementation would use a binary search rather than a linear search here.[1] However, this code is currently only called at scene initialization time, so the lack of these optimizations doesn't currently impact rendering performance.

⟨*Advance to first relevant wavelength segment*⟩ ≡ 268
```
int i = 0;
while (lambdaStart > lambda[i+1]) ++i;
```

The loop below iterates over each of the linear segments that the averaging range overlaps. For each one, the `SEG_AVG()` macro computes the average value over the wavelength range `segStart` to `segEnd` by averaging the values of the function at those two points. The values in turn are computed by `INTERP()`, which linearly interpolates between the two end points at the given wavelength.

The `min()` and `max()` calls below compute the wavelength range to average over within the segment; note that they naturally handle the cases where `lambdaStart`, `lambdaEnd`, or both of them are within the current segment.

⟨*Loop over wavelength sample segments and add contributions*⟩ ≡ 268
```
#define INTERP(w, i) \
    Lerp(((w) - lambda[i]) / (lambda[(i)+1] - lambda[i]), \
         vals[i], vals[(i)+1])
#define SEG_AVG(wl0, wl1, i) (0.5f * (INTERP(wl0, i) + INTERP(wl1, i)))
for (; i+1 < n && lambdaEnd >= lambda[i]; ++i) {
    float segStart = max(lambdaStart, lambda[i]);
    float segEnd =   min(lambdaEnd,   lambda[i+1]);
    sum += SEG_AVG(segStart, segEnd, i) * (segEnd - segStart);
}
#undef INTERP
#undef SEG_AVG
```

1 An even more efficient implementation would take advantage of the fact that the calling code will generally ask for interpolated values over a series of adjacent wavelength ranges, and possibly take all of the ranges in a single call. It could then incrementally find the starting index for the next interpolation starting from the end of the previous one.

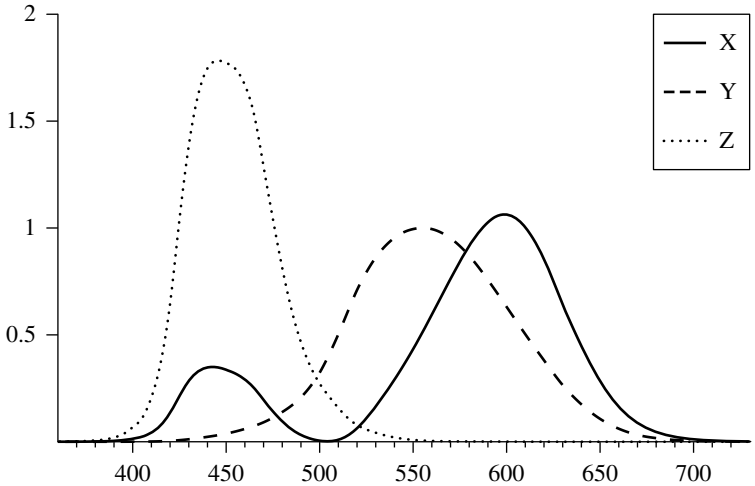


Figure 5.3: Computing the XYZ Values for an Arbitrary SPD. The SPD is convolved with each of the three matching curves to compute the values x_λ, y_λ, and z_λ, using Equation (5.1).

5.2.1 XYZ COLOR

A remarkable property of the human visual system makes it possible to represent colors for human perception with just three floating-point numbers. The *tristimulus theory* of color perception says that all visible SPDs can be accurately represented for human observers with three values, x_λ, y_λ, and z_λ. Given an SPD $S(\lambda)$, these values are computed by integrating its product with the *spectral matching curves* $X(\lambda)$, $Y(\lambda)$, and $Z(\lambda)$:

$$\begin{aligned} x_\lambda &= \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \int_\lambda S(\lambda)\, X(\lambda)\, \mathrm{d}\lambda \\ y_\lambda &= \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \int_\lambda S(\lambda)\, Y(\lambda)\, \mathrm{d}\lambda \\ z_\lambda &= \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \int_\lambda S(\lambda)\, Z(\lambda)\, \mathrm{d}\lambda. \end{aligned} \tag{5.1}$$

These curves were determined by the Commission Internationale de l'Éclairage (CIE) standards body after a series of experiments with human test subjects and are graphed in Figure 5.3. It is believed that these matching curves are generally similar to the responses of the three types of color-sensitive cones in the human retina. Remarkably, SPDs with substantially different distributions may have very similar x_λ, y_λ, and z_λ values. To the human observer, such SPDs actually appear the same visually. Pairs of such spectra are called *metamers*.

This brings us to a subtle point about representations of spectral power distributions. Most color spaces attempt to model colors that are visible to humans, and therefore use

only three coefficients, exploiting the tristimulus theory of color perception. Although XYZ works well to represent a given SPD to be displayed for a human observer, it is *not* a particularly good set of basis functions for spectral computation. For example, although XYZ values would work well to describe the perceived color of lemon skin or a fluorescent light individually (recall Figure 5.1), the product of their respective XYZ values is likely to give a noticeably different XYZ color than the XYZ value computed by multiplying more accurate representations of their SPDs and *then* computing the XYZ value.

pbrt provides the values of the standard $X(\lambda)$, $Y(\lambda)$, and $Z(\lambda)$ response curves sampled at 1 nm increments from 360 nm to 830 nm. The wavelengths of the ith sample in the arrays below are given by the ith element of CIE_lambda; having the wavelengths of the samples explicitly represented in this way makes it easy to pass the XYZ samples into functions like AverageSpectrumSamples() that take such an array of wavelengths as a parameter.

⟨*Spectral Data Declarations*⟩ ≡
```
static const int nCIESamples = 471;
extern const float CIE_X[nCIESamples];
extern const float CIE_Y[nCIESamples];
extern const float CIE_Z[nCIESamples];
extern const float CIE_lambda[nCIESamples];
```

SampledSpectrum uses these samples to compute the XYZ matching curves in its spectral representation (i.e., themselves as SampledSpectrums). The yint value stores the integral of the Y matching curve over the wavelength range; this value is needed for some conversions between full spectra and x_λ, y_λ, z_λ coefficients.

⟨*SampledSpectrum Private Data*⟩ ≡ 266
```
static SampledSpectrum X, Y, Z;
static float yint;
```

The SampledSpectrum XYZ matching curves are computed in the SampledSpectrum::Init() method, which is called at system startup time by the pbrtInit() function defined in Section B.2.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
static void Init() {
    ⟨Compute XYZ matching functions for SampledSpectrum 272⟩
    ⟨Compute RGB to spectrum functions for SampledSpectrum⟩
}
```

AverageSpectrumSamples() 268
nCIESamples 271
pbrtInit() 1051
SampledSpectrum 266

Given the wavelength range and number of samples for SampledSpectrum, computing the values of the matching functions for each sample is just a matter of computing the sample's wavelength range and using the AverageSpectrumSamples() routine.

⟨*Compute XYZ matching functions for* SampledSpectrum⟩ ≡ 271
```
for (int i = 0; i < nSpectralSamples; ++i) {
    float wl0 = Lerp(float(i) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
    float wl1 = Lerp(float(i+1) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
    X.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_X, nCIESamples,
                                    wl0, wl1);
    Y.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Y, nCIESamples,
                                    wl0, wl1);
    Z.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Z, nCIESamples,
                                    wl0, wl1);
    yint += Y.c[i];
}
```

All Spectrum implementations in pbrt must provide a method that converts their SPD to $(x_\lambda, y_\lambda, z_\lambda)$ coefficients. This method is called, for example, in the process of updating pixels in the image. When a Spectrum is provided to the ImageFilm, the ImageFilm converts the SPD into XYZ coefficients as a first step in the process of finally turning them into RGB values used for storage and/or display. The implementation for SampledSpectrum computes the integrals from Equation (5.1) with a Riemann sum:

$$x_\lambda \approx \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \sum_i X_i c_i,$$

and so forth.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
void ToXYZ(float xyz[3]) const {
    xyz[0] = xyz[1] = xyz[2] = 0.f;
    for (int i = 0; i < nSpectralSamples; ++i) {
        xyz[0] += X.c[i] * c[i];
        xyz[1] += Y.c[i] * c[i];
        xyz[2] += Z.c[i] * c[i];
    }
    xyz[0] /= yint;
    xyz[1] /= yint;
    xyz[2] /= yint;
}
```

The y coordinate of XYZ color is closely related to *luminance*, which measures the perceived brightness of a color. Luminance is described in more detail in Section 5.4.3. We provide a method to compute y alone in a separate method as often only the luminance of a spectrum is desired. (For example, some of the light transport algorithms in Chapters 15–17 use luminance as a measure of relative importance of light-carrying paths through the scene.)

AverageSpectrumSamples() 268
CIE_lambda 271
CIE_X 271
CIE_Y 271
CIE_Z 271
ImageFilm 404
Lerp() 1000
nCIESamples 271
nSpectralSamples 266
sampledLambdaEnd 266
sampledLambdaStart 266
Spectrum 263

⟨*SampledSpectrum Public Methods*⟩ +≡ 266

```
float y() const {
    float yy = 0.f;
    for (int i = 0; i < nSpectralSamples; ++i)
        yy += Y.c[i] * c[i];
    return yy / yint;
}
```

5.2.2 RGB COLOR

When we display an RGB color on a display, the spectrum that is actually displayed is basically determined by the weighted sum of three spectral response curves, one for each of red, green, and blue, as emitted by the display's phosphors, LED or LCD elements, or plasma cells.[2] Figure 5.4 plots the red, green, and blue distributions emitted by a LED display and a LCD display; note that they are remarkably different. Figure 5.5 in turn shows the SPDs that result from displaying the RGB color (0.6, 0.3, 0.2) on those displays. Not surprisingly, the resulting SPDs are quite different as well. This example illustrates that using RGB values provided by the user to describe a particular color is actually only meaningful given knowledge of the characteristics of the display they were using when they selected the RGB values.

Given an $(x_\lambda, y_\lambda, z_\lambda)$ representation of an SPD, we can convert it to corresponding RGB coefficients, given the choice of a particular set of SPDs that define red, green, and blue for a display of interest. The conversion routines implemented in `pbrt` are based on a standard set of these RGB spectra that has been defined for high-definition television. Given the spectral response curves $R(\lambda)$, $G(\lambda)$, and $B(\lambda)$, for a particular display, RGB coefficients can be computed by integrating the reponse curves with the SPD:

$$\begin{aligned} r &= \int R(\lambda)S(\lambda)\mathrm{d}\lambda = \int R(\lambda)(x_\lambda X(\lambda) + y_\lambda Y(\lambda) + z_\lambda Z(\lambda))\,\mathrm{d}\lambda \\ &= x_\lambda \int R(\lambda)X(\lambda)\,\mathrm{d}\lambda + y_\lambda \int R(\lambda)Y(\lambda)\,\mathrm{d}\lambda + z_\lambda \int R(\lambda)Z(\lambda)\,\mathrm{d}\lambda. \end{aligned}$$

The integrals of the products of $R(\lambda)X(\lambda)$ and so forth can be precomputed for given response curves, making it possible to express the full conversion as a matrix:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{pmatrix} \int R(\lambda)X(\lambda)\mathrm{d}\lambda & \int R(\lambda)Y(\lambda)\mathrm{d}\lambda & \int R(\lambda)Z(\lambda)\mathrm{d}\lambda \\ \int G(\lambda)X(\lambda)\mathrm{d}\lambda & \int G(\lambda)Y(\lambda)\mathrm{d}\lambda & \int G(\lambda)Z(\lambda)\mathrm{d}\lambda \\ \int B(\lambda)X(\lambda)\mathrm{d}\lambda & \int B(\lambda)Y(\lambda)\mathrm{d}\lambda & \int B(\lambda)Z(\lambda)\mathrm{d}\lambda \end{pmatrix} \begin{bmatrix} x_\lambda \\ y_\lambda \\ z_\lambda \end{bmatrix}.$$

⟨*Spectrum Utility Declarations*⟩ +≡

```
inline void XYZToRGB(const float xyz[3], float rgb[3]) {
    rgb[0] =  3.240479f*xyz[0] - 1.537150f*xyz[1] - 0.498535f*xyz[2];
    rgb[1] = -0.969256f*xyz[0] + 1.875991f*xyz[1] + 0.041556f*xyz[2];
    rgb[2] =  0.055648f*xyz[0] - 0.204043f*xyz[1] + 1.057311f*xyz[2];
}
```

CoefficientSpectrum::c 264
nSpectralSamples 266
SampledSpectrum::Y 271

2 This model is admittedly a simplification in that it neglects any additional processing the display does; in particular, many displays perform nonlinear remappings of the displayed values.

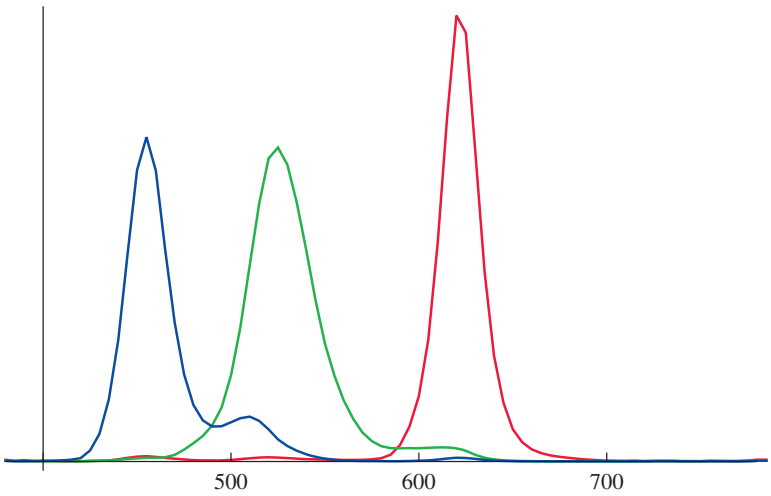


Figure 5.4: Red, Green, and Blue Emission Curves for an LCD Display and a LED Display. The top plot shows the curves for an LCD display and the bottom shows them for an LED. These two displays have quite different emission profiles. *(Data courtesy of X-Rite, Inc.)*

The inverse of this matrix gives coefficients to convert given RGB values, expressed with respect to a particular set of RGB response curves, to $(x_\lambda, y_\lambda, z_\lambda)$ coefficients.

⟨*Spectrum Utility Declarations*⟩ +≡

```
inline void RGBToXYZ(const float rgb[3], float xyz[3]) {
    xyz[0] = 0.412453f*rgb[0] + 0.357580f*rgb[1] + 0.180423f*rgb[2];
    xyz[1] = 0.212671f*rgb[0] + 0.715160f*rgb[1] + 0.072169f*rgb[2];
    xyz[2] = 0.019334f*rgb[0] + 0.119193f*rgb[1] + 0.950227f*rgb[2];
}
```

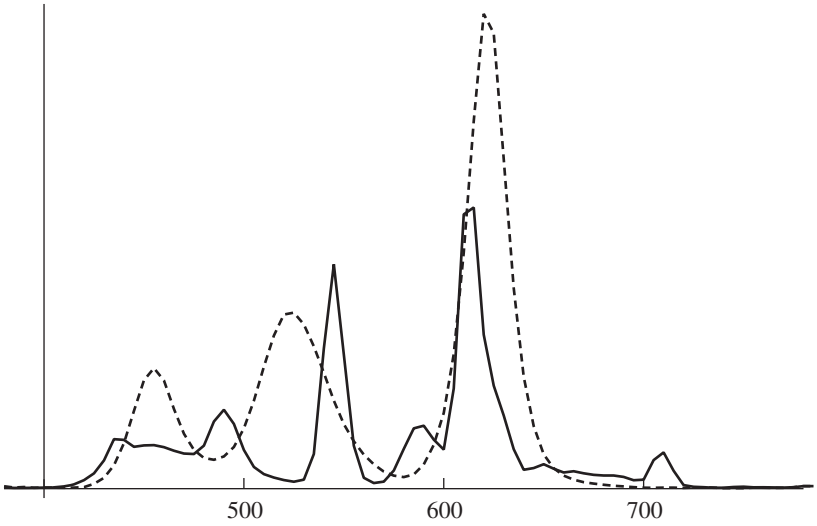


Figure 5.5: SPDs from Displaying the RGB Color (0.6, 0.3, 0.2) **on LED and LCD Displays.** The resulting emitted SPDs are remarkably different, even given the same RGB values, due to the different emission curves illustrated in Figure 5.4.

Given these functions, a `SampledSpectrum` can convert to RGB coefficients by first converting to XYZ and then using the `XYZToRGB()` utility function.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
void ToRGB(float rgb[3]) const {
    float xyz[3];
    ToXYZ(xyz);
    XYZToRGB(xyz, rgb);
}
```

An `RGBSpectrum` can also be created easily, using the `ToRGB()` method.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
RGBSpectrum ToRGBSpectrum() const;
```

Going the other way and converting from RGB or XYZ values to a SPD isn't as easy: the problem is highly under-constrained. Recall that an infinite number of different SPDs have the same $(x_\lambda, y_\lambda, z_\lambda)$ (and thus, RGB) coefficients. Thus, given an RGB or $(x_\lambda, y_\lambda, z_\lambda)$ value, there are an infinite number of possible SPDs that could be chosen for it. There are a number of desirable criteria that we'd like a conversion function to have:

RGBSpectrum 279
SampledSpectrum 266
SampledSpectrum::ToXYZ() 272
XYZToRGB() 273

- If all of the RGB coefficients have the same value, the resulting SPD should be constant.
- In general, it's desirable that the computed SPD be smooth. Most real-world objects have relatively smooth spectra. (The main source of spiky spectra is light sources,

especially fluorescents. Fortunately, actual spectral data is more commonly available for illuminants than it is for reflectances.)

The smoothness goal is one of the reasons why constructing a SPD as a weighted sum of a display's $R(\lambda)$, $G(\lambda)$, and $B(\lambda)$ SPDs is not a good solution: as shown in Figure 5.4, those functions are generally irregular and spiky, and a weighted sum of them will thus not be a very smooth SPD. Although the result will be a metamer of the given RGB values, it's likely not an accurate representation of the SPD of the actual object.

Here we implement a method for converting RGBs to SPDs suggested by Smits (1999) that tries to achieve the goals above. This approach is based on the observation that a good start is to compute individual SPDs for red, green, and blue that are smooth and such that computing the weighted sum of them with the given RGB coefficients and then converting back to RGB give a result that is close to the original RGB coefficients. He found such spectra through a numerical optimization procedure.

Smits observed that two additional improvements could be made to this basic approach. First, rather than representing constant spectra by the sums of the computed red, green, and blue SPDs, the sum of which isn't exactly constant, it's better to represent constant spectra with constant SPDs. Second, mixtures of colors like yellow (a mixture of red and green) that are a mixture of two of the primaries are better represented by their own precomputed smooth SPDs rather than the sum of SPDs for the two corresponding primaries.

The following arrays store SPDs that meet these criteria, with their samples' wavelengths in `RGB2SpectLambda[]` (these data were generated by Karl vom Berge).

⟨*Spectral Data Declarations*⟩ +≡
```
static const int nRGB2SpectSamples = 32;
extern const float RGB2SpectLambda[nRGB2SpectSamples];
extern const float RGBRefl2SpectWhite[nRGB2SpectSamples];
extern const float RGBRefl2SpectCyan[nRGB2SpectSamples];
extern const float RGBRefl2SpectMagenta[nRGB2SpectSamples];
extern const float RGBRefl2SpectYellow[nRGB2SpectSamples];
extern const float RGBRefl2SpectRed[nRGB2SpectSamples];
extern const float RGBRefl2SpectGreen[nRGB2SpectSamples];
extern const float RGBRefl2SpectBlue[nRGB2SpectSamples];
```

If a given RGB color describes illumination from a light source, better results are achieved if the conversion tables are computed using the spectral power distribution of a representative illumination source to define "white" rather than using a constant spectrum as they are for the tables above that are used for reflectances. The `RGBIllum2Spect` arrays use the D65 spectral power distribution, which has been standardized by the CIE to represent midday sunlight.

⟨*Spectral Data Declarations*⟩ +≡
```
extern const float RGBIllum2SpectWhite[nRGB2SpectSamples];
extern const float RGBIllum2SpectCyan[nRGB2SpectSamples];
extern const float RGBIllum2SpectMagenta[nRGB2SpectSamples];
```
nRGB2SpectSamples 276

```
extern const float RGBIllum2SpectYellow[nRGB2SpectSamples];
extern const float RGBIllum2SpectRed[nRGB2SpectSamples];
extern const float RGBIllum2SpectGreen[nRGB2SpectSamples];
extern const float RGBIllum2SpectBlue[nRGB2SpectSamples];
```

The fragment ⟨*Compute RGB to spectrum functions for* SampledSpectrum⟩, which is called from SampledSpectrum::Init(), isn't included here; it initializes the following SampledSpectrum values by resampling the RGBRefl2Spect and RGBIllum2Spect distributions using the AverageSpectrumSamples() function.

⟨*SampledSpectrum Private Data*⟩ +≡ 266

```
static SampledSpectrum rgbRefl2SpectWhite, rgbRefl2SpectCyan;
static SampledSpectrum rgbRefl2SpectMagenta, rgbRefl2SpectYellow;
static SampledSpectrum rgbRefl2SpectRed, rgbRefl2SpectGreen;
static SampledSpectrum rgbRefl2SpectBlue;
```

⟨*SampledSpectrum Private Data*⟩ +≡ 266

```
static SampledSpectrum rgbIllum2SpectWhite, rgbIllum2SpectCyan;
static SampledSpectrum rgbIllum2SpectMagenta, rgbIllum2SpectYellow;
static SampledSpectrum rgbIllum2SpectRed, rgbIllum2SpectGreen;
static SampledSpectrum rgbIllum2SpectBlue;
```

The SampledSpectrum::FromRGB() method converts from the given RGB values to a full SPD. In addition to the RGB values, it takes an enumerant value that denotes whether the RGB value represents surface reflectance or an illuminant; the corresponding rgbIllum2Spect or rgbRefl2Spect values are used for the conversion.

⟨*Spectrum Utility Declarations*⟩ +≡

```
enum SpectrumType { SPECTRUM_REFLECTANCE, SPECTRUM_ILLUMINANT };
```

⟨*Spectrum Method Definitions*⟩ +≡

```
SampledSpectrum SampledSpectrum::FromRGB(const float rgb[3],
                                         SpectrumType type) {
    SampledSpectrum r;
    if (type == SPECTRUM_REFLECTANCE) {
        ⟨Convert reflectance spectrum to RGB 278⟩
    }
    else {
        ⟨Convert illuminant spectrum to RGB⟩
    }
    return r.Clamp();
}
```

AverageSpectrumSamples() 268
SampledSpectrum 266
SampledSpectrum::Init() 271
Spectrum::Clamp() 265
SpectrumType 277
SPECTRUM_REFLECTANCE 277

Here we'll show the conversion process for reflectances. The computation for illuminants is the same, just using the different conversion values. First, the implementation determines whether the red, green, or blue channel is the smallest.

⟨*Convert reflectance spectrum to RGB*⟩ ≡ 277
```
if (rgb[0] <= rgb[1] && rgb[0] <= rgb[2]) {
    ⟨Compute reflectance SampledSpectrum with rgb[0] as minimum 278⟩
}
else if (rgb[1] <= rgb[0] && rgb[1] <= rgb[2]) {
    ⟨Compute reflectance SampledSpectrum with rgb[1] as minimum⟩
}
else {
    ⟨Compute reflectance SampledSpectrum with rgb[2] as minimum⟩
}
```

Here is the code for the case of a red component being the smallest. (The cases for green and blue are analogous and not included in the book here.) If red is the smallest, we know that green and blue have greater values than red. As such, we can start to convert the final SPD to return by assigning to it the value of the red component times the white spectrum in `rgbRefl2SpectWhite`. Having done this, the remaining RGB value left to process is $(0, g - r, b - r)$. The code in turn determines which of the remaining two components is the smallest. This value, times the cyan (green and blue) spectrum, is added to the result and we're left with either $(0, g - b, 0)$ or $(0, 0, b - g)$. Based on whether the green or blue channel is non-zero, the green or blue SPD is scaled by the remainder and the conversion is complete.

⟨*Compute reflectance* `SampledSpectrum` *with* `rgb[0]` *as minimum*⟩ ≡ 278
```
r += rgb[0] * rgbRefl2SpectWhite;
if (rgb[1] <= rgb[2]) {
    r += (rgb[1] - rgb[0]) * rgbRefl2SpectCyan;
    r += (rgb[2] - rgb[1]) * rgbRefl2SpectBlue;
}
else {
    r += (rgb[2] - rgb[0]) * rgbRefl2SpectCyan;
    r += (rgb[1] - rgb[2]) * rgbRefl2SpectGreen;
}
```

Given the method to convert from RGB, converting from XYZ color is easy. We first convert from XYZ to RGB and then use the `FromRGB()` method.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
static SampledSpectrum FromXYZ(const float xyz[3],
        SpectrumType type = SPECTRUM_REFLECTANCE) {
    float rgb[3];
    XYZToRGB(xyz, rgb);
    return FromRGB(rgb, type);
}
```

Finally, we provide a constructor that converts from an instance of the `RGBSpectrum` class, again using the infrastructure above.

RGBSpectrum 279
SampledSpectrum 266
SampledSpectrum::
FromRGB() 277
SampledSpectrum::
rgbRefl2SpectBlue 277
SampledSpectrum::
rgbRefl2SpectCyan 277
SampledSpectrum::
rgbRefl2SpectGreen 277
SampledSpectrum::
rgbRefl2SpectWhite 277
SpectrumType 277
SPECTRUM_REFLECTANCE 277
XYZToRGB() 273

⟨*Spectrum Method Definitions*⟩ +≡
```
SampledSpectrum::SampledSpectrum(const RGBSpectrum &r, SpectrumType t) {
    float rgb[3];
    r.ToRGB(rgb);
    *this = SampledSpectrum::FromRGB(rgb, t);
}
```

5.3 RGBSpectrum IMPLEMENTATION

The RGBSpectrum implementation here represents SPDs with a weighted sum of red, green, and blue components. Recall that this representation is ill defined: given two different computer displays, having them display the same RGB value won't cause them to emit the same SPD. Thus, in order for a set of RGB values to specify an actual SPD, we must know the monitor primaries that they are defined in terms of; this information is generally not provided along with RGB values.

The RGB representation is nevertheless convenient: almost all 3D modeling and design tools use RGB colors, and most 3D content is specified in terms of RGB. Furthermore, it's computationally and storage efficient, requiring just three floating-point values to represent. Our implementation of RGBSpectrum inherits from CoefficientSpectrum, specifying three components to store. Thus, all of the arithmetic operations defined earlier are automatically available for the RGBSpectrum.

⟨*Spectrum Declarations*⟩ +≡
```
class RGBSpectrum : public CoefficientSpectrum<3> {
public:
    ⟨RGBSpectrum Public Methods 279⟩
};
```

⟨*RGBSpectrum Public Methods*⟩ ≡ 279
```
RGBSpectrum(float v = 0.f) : CoefficientSpectrum<3>(v) { }
RGBSpectrum(const CoefficientSpectrum<3> &v)
    : CoefficientSpectrum<3>(v) { }
```

Beyond the basic arithmetic operators, the RGBSpectrum needs to provide methods to convert to and from XYZ and RGB representations. For the RGBSpectrum these are trivial. Note that FromRGB() takes a SpectrumType parameter like the SampledSpectrum instance of this method. Although it's unused here, the FromRGB() methods of these two classes must have matching signatures so that the rest of the system can call them consistently regardless of which spectral representation is being used.

CoefficientSpectrum 264
CoefficientSpectrum::c 264
RGBSpectrum 279
RGBSpectrum::ToRGB() 280
SampledSpectrum 266
SampledSpectrum::FromRGB() 277
SpectrumType 277
SPECTRUM_REFLECTANCE 277

⟨*RGBSpectrum Public Methods*⟩ +≡ 279
```
static RGBSpectrum FromRGB(const float rgb[3],
        SpectrumType type = SPECTRUM_REFLECTANCE) {
    RGBSpectrum s;
    s.c[0] = rgb[0];
    s.c[1] = rgb[1];
    s.c[2] = rgb[2];
    return s;
}
```

Similarly, spectrum representations must be able to convert themselves to RGB values. For the RGBSpectrum, the implementation can sidestep the question of what particular RGB primaries are used to represent the spectral distribution and just return the RGB coefficients directly, assuming that the primaries are the same as the ones already being used to represent the color.

⟨*RGBSpectrum Public Methods*⟩ +≡ 279
```
void ToRGB(float *rgb) const {
    rgb[0] = c[0];
    rgb[1] = c[1];
    rgb[2] = c[2];
}
```

All spectrum representations must also be able to convert themselves to an RGBSpectrum object. This is again trivial here.

⟨*RGBSpectrum Public Methods*⟩ +≡ 279
```
const RGBSpectrum &ToRGBSpectrum() const {
    return *this;
}
```

The implementations of the RGBSpectrum::ToXYZ(), RGBSpectrum::FromXYZ(), and RGBSpectrum::y() methods are based on the utility functions defined above and are not included here.

To create an RGB spectrum from an arbitrary sampled SPD, FromSampled() converts the spectrum to XYZ and then to RGB. It evaluates the piecewise linear sampled spectrum at 1-nm steps, using the InterpolateSpectrumSamples() utility function, at each of the wavelengths where there is a value for the CIE matching functions. It then uses this value to compute the Riemann sum to approximate the XYZ integrals.

⟨*RGBSpectrum Public Methods*⟩ +≡ 279
```
static RGBSpectrum FromSampled(const float *lambda, const float *v,
                               int n) {
    ⟨Sort samples if unordered, use sorted for returned spectrum 267⟩
    float xyz[3] = { 0, 0, 0 };
    float yint = 0.f;
    for (int i = 0; i < nCIESamples; ++i) {
        yint += CIE_Y[i];
        float val = InterpolateSpectrumSamples(lambda, v, n,
                                               CIE_lambda[i]);
        xyz[0] += val * CIE_X[i];
        xyz[1] += val * CIE_Y[i];
        xyz[2] += val * CIE_Z[i];
    }
    xyz[0] /= yint;
    xyz[1] /= yint;
    xyz[2] /= yint;
    return FromXYZ(xyz);
}
```

CIE_lambda 271
CIE_X 271
CIE_Y 271
CIE_Z 271
CoefficientSpectrum::c 264
InterpolateSpectrum Samples() 281
nCIESamples 271
RGBSpectrum 279
RGBSpectrum::FromXYZ() 280

`InterpolateSpectrumSamples()` takes a possibly irregularly sampled set of wavelengths and SPD values (λ_i, v_i) and returns the value of the SPD at the given wavelength λ, linearly interpolating between the two sample values that bracket λ.

⟨*Spectrum Method Definitions*⟩ +≡
```
float InterpolateSpectrumSamples(const float *lambda, const float *vals,
                                 int n, float l) {
    if (l <= lambda[0])   return vals[0];
    if (l >= lambda[n-1]) return vals[n-1];
    for (int i = 0; i < n-1; ++i) {
        if (l >= lambda[i] && l <= lambda[i+1]) {
            float t = (l - lambda[i]) / (lambda[i+1] - lambda[i]);
            return Lerp(t, vals[i], vals[i+1]);
        }
    }
}
```

5.4 BASIC RADIOMETRY

Radiometry provides a set of ideas and mathematical tools to describe light propagation and reflection. It forms the basis of the derivation of the rendering algorithms that will be used throughout the rest of this book. Interestingly enough, radiometry wasn't originally derived from first principles using the physics of light, but was built on an abstraction of light based on particles flowing through space. As such, effects like polarization of light do not naturally fit into this framework, although connections have since been made between radiometry and Maxwell's equations, giving radiometry a solid basis in physics.

Radiative transfer is the phenomenological study of the transfer of radiant energy. It is based on radiometric principles and operates at the *geometric optics* level, where macroscopic properties of light suffice to describe how light interacts with objects much larger than the light's wavelength. It is not uncommon to incorporate phenomena from wave optics models of light, but these results need to be expressed in the language of radiative transfer's basic abstractions.[3] In this manner, it is possible to describe interactions of light with objects of approximately the same size as the wavelength of the light, and thereby model effects like dispersion and interference. At an even finer level of detail, quantum mechanics is needed to describe light's interaction with atoms. Fortunately, direct simulation of quantum mechanical principles is unnecessary for solving rendering problems in computer graphics, so the intractability of such an approach is avoided.

In `pbrt`, we will assume that geometric optics is an adequate model for the description of light and light scattering. This leads to a few basic assumptions about the behavior of light that will be used implicitly throughout the system:

3 Preisendorfer (1965) has connected radiative transfer theory to Maxwell's classical equations describing electromagnetic fields. His framework both demonstrates their equivalence and makes it easier to apply results from one worldview to the other. More recent work was done in this area by Fante (1981).

- *Linearity:* The combined effect of two inputs to an optical system is always equal to the sum of the effects of each of the inputs individually.
- *Energy conservation:* When light scatters from a surface or from participating media, the scattering events can never produce more energy than they started with.
- *No polarization:* We will ignore polarization of the electromagnetic field; therefore, the only relevant property of light is its distribution by wavelength (or, equivalently, frequency).
- *No fluorescence or phosphorescence:* The behavior of light at one wavelength is completely independent of light's behavior at other wavelengths or times. As with polarization, it is not too difficult to include these effects, but they would add relatively little practical value to the system.
- *Steady state:* Light in the environment is assumed to have reached equilibrium, so its radiance distribution isn't changing over time. This happens nearly instantaneously with light in realistic scenes, so it is not a limitation in practice. Note that phosphorescence also violates the steady-state assumption.

The most significant loss from adopting a geometric optics model is that diffraction and interference effects cannot easily be accounted for. As noted by Preisendorfer (1965), this is a hard problem to fix because, for example, the total flux over two areas isn't necessarily equal to the sum of flux over each individual area in the presence of those effects (p. 24).

5.4.1 BASIC QUANTITIES

There are four radiometric quantities that are central to rendering: flux, irradiance/radiant exitance, intensity, and radiance. All of these quantities are generally wavelength dependent. For the remainder of this chapter, we will not make this dependence explicit, but this property is important to keep in mind.

Flux

Radiant flux, also known as *power*, is the total amount of energy passing through a surface or region of space per unit time. Its units are joules/second (J/s), or more commonly watts (W), and it is normally denoted by the symbol Φ. Total emission from light sources is generally described in terms of flux. Figure 5.6 shows flux from a point light source measured by the total amount of energy passing through an imaginary sphere around the light. Note that the total amount of flux measured on either of the two spheres in Figure 5.6 is the same—although less energy is passing through any local part of the large sphere than the small sphere, the greater area of the large sphere means that the total flux is the same.

Irradiance and Radiant Exitance

Irradiance (E) is the area density of flux arriving at a surface, and *radiant exitance* (M) is the area density of flux leaving a surface. These measurements have units of W/m^2. (The term "irradiance" is sometimes also used to refer to flux leaving a surface, but for clarity we'll use different terms for the two cases.)

For the point light source example in Figure 5.6, irradiance at a point on the outer sphere is less than the irradiance at a point on the inner sphere, since the surface area of the outer

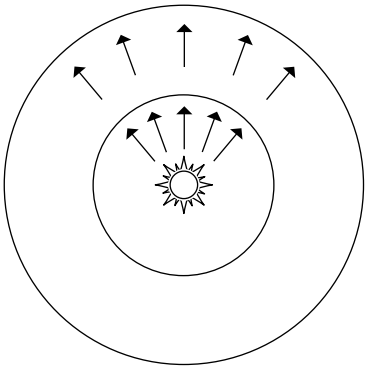


Figure 5.6: Radiant flux, Φ, measures energy passing through a surface or region of space. Here, flux from a point light source is measured at spheres that surround the light.

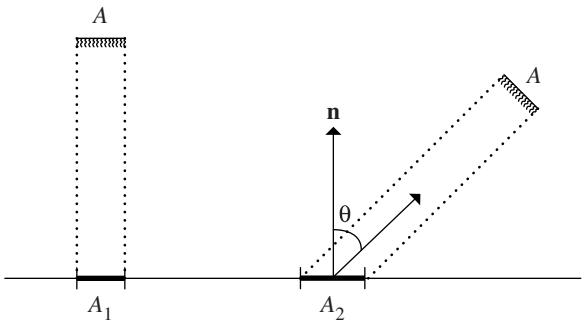


Figure 5.7: Lambert's Law. Irradiance (E) arriving at a surface varies according to the cosine of the angle of incidence of illumination, since illumination is over a larger area at smaller incident angles.

sphere is larger. In particular, for a sphere in this configuration that has radius r,

$$E = \frac{\Phi}{4\pi r^2}.$$

This fact explains why the amount of energy received from a light falls off with the squared distance from the light.

The irradiance equation can also help us understand the origin of *Lambert's law*, which says that the amount of light arriving at a surface is proportional to the cosine of the angle between the light direction and the surface normal (Figure 5.7). Consider a light source with area A and flux Φ that is illuminating a surface. If the light is shining directly down

on the surface (as on the left side of the figure), then the area on the surface receiving light A_1 is equal to A. Irradiance at any point inside A_1 is then

$$E_1 = \frac{\Phi}{A}.$$

However, if the light is at an angle to the surface, the area on the surface receiving light is larger. If A is small, then the area receiving flux, A_2, is roughly $A/\cos\theta$. For points inside A_2, the irradiance is therefore

$$E_2 = \frac{\Phi \cos\theta}{A}.$$

More formally, to cover the general case where the emitted flux distribution is not constant, irradiance at a point is defined as

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A},$$

where the differential flux from the light is computed over the differential area receiving flux.

Solid Angle and Intensity

In order to define *intensity*, we first need to define the notion of a *solid angle*. Solid angles are just the extension of two-dimensional angles in a plane to an angle on a sphere. The *planar angle* is the total angle subtended by some object with respect to some position (Figure 5.8). Consider the unit circle around the point p; if we project the shaded object onto that circle, some length of the circle s will be covered by its projection. The arc length of s (which is the same as the angle θ) is the angle subtended by the object. Planar angles are measured in *radians*.

The solid angle extends the 2D unit circle to a 3D unit sphere (Figure 5.9). The total area s is the solid angle subtended by the object. Solid angles are measured in *steradians*. The entire sphere subtends a solid angle of 4π, and a hemisphere subtends 2π.

The set of points on the unit sphere centered at a point p can be used to describe the vectors anchored at p. We will frequently use the symbol ω to indicate these directions, and we will use the convention that ω is a normalized vector.

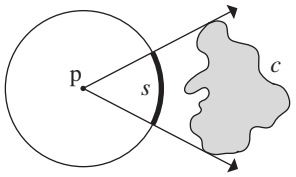


Figure 5.8: Planar Angle. The planar angle of an object c as seen from a point p is equal to the angle it subtends as seen from p, or equivalently as the length of the arc s on the unit sphere.

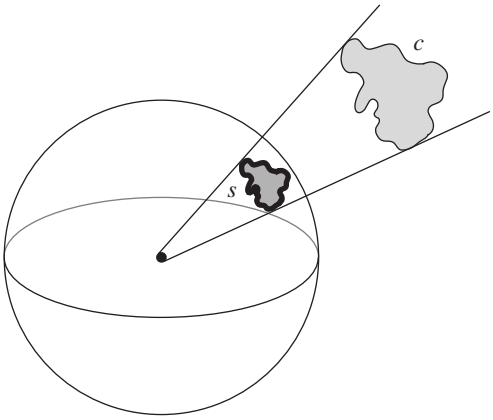


Figure 5.9: Solid Angle. The solid angle s subtended by an object c in three dimensions is computed by projecting c onto the unit sphere and measuring the area of its projection.

Given these definitions, we can now define intensity, which is flux density per solid angle:

$$I = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega}.$$

Intensity describes the directional distribution of light, but it is only meaningful for point light sources.

Radiance

The final, and most important, radiometric quantity is *radiance*, L. Radiance is the flux density per unit area, per unit solid angle. In terms of flux, it is

$$L = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega\, \mathrm{d}A^{\perp}}, \tag{5.2}$$

where $\mathrm{d}A^{\perp}$ is the projected area of $\mathrm{d}A$ on a hypothetical surface perpendicular to ω (Figure 5.10). Thus, it is the limit of the measurement of incident light at the surface as a cone of incident directions of interest $\mathrm{d}\omega$ becomes very small, and as the local area of interest on the surface $\mathrm{d}A$ also becomes very small.

Of all of these radiometric quantities, radiance will be the one used most frequently throughout the rest of the book. An intuitive reason for this is that in some sense it's the most fundamental of all the radiometric quantities; if radiance is given, then all of the other values can be computed in terms of integrals of radiance over areas and directions. Another nice property of radiance is that it remains constant along rays through empty space. It is thus a natural quantity to compute with ray tracing.

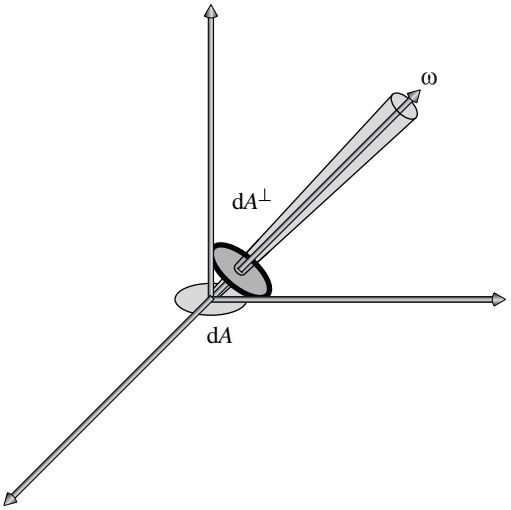


Figure 5.10: Radiance L is defined as flux per unit solid angle $d\omega$ per unit projected area $dA^{\perp}$.

5.4.2 INCIDENT AND EXITANT RADIANCE FUNCTIONS

When discussing radiance at a point in an environment and when writing equations based on radiance, it is useful to make a distinction between radiance arriving at the point (for example, due to illumination from a light source) and radiance leaving that point (for example, due to reflection from a surface). In `pbrt` we will use the idea of incident and exitant radiance functions to distinguish between these two cases. The approach used here is based on Section 3.5 of Veach (1997).

Consider a point p on the surface of an object. There is some distribution of radiance arriving at the point that can be described mathematically by a function of position and direction. This function is denoted by $L_i(p, \omega)$ (Figure 5.11). The function that describes the outgoing reflected radiance from the surface at that point is denoted by $L_o(p, \omega)$. Note that in both cases the direction vector ω is oriented to point away from p, but be aware that some authors use a notation where ω is reversed for L_i terms so that it points toward p.

It is important to see that, in general,

$$L_i(p, \omega) \neq L_o(p, \omega).$$

Another property to keep in mind is that at a point in space where there is no surface and where there is no participating media causing scattering, these functions are related by

$$L_o(p, \omega) = L_i(p, -\omega).$$

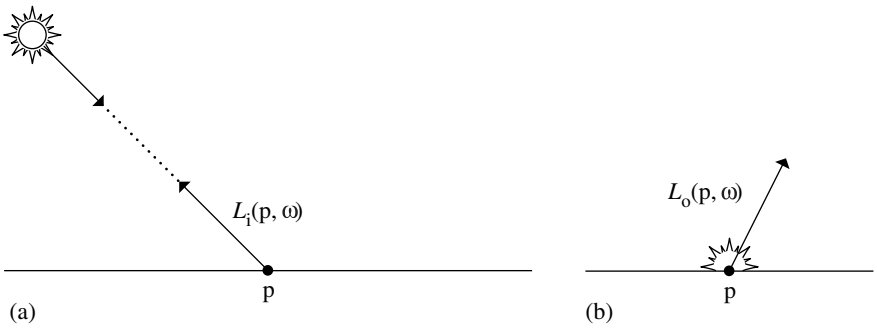


Figure 5.11: (a) The incident radiance function $L_i(p, \omega)$ describes the distribution of radiance as a function of position and direction. (b) The exitant radiance function $L_o(p, \omega)$ gives the distribution of radiance leaving the surface. Note that for both functions, ω is oriented to point away from the surface and, thus for example, $L_i(p, -\omega)$ gives the radiance arriving on the other side of the surface than the one where ω lies.

5.4.3 LUMINANCE AND PHOTOMETRY

All of the radiometric measurements like flux, radiance, and so forth have corresponding photometric measurements. *Photometry* is the study of visible electromagnetic radiation in terms of its perception by the human visual system. Each spectral radiometric quantity can be converted to its corresponding photometric quantity by integrating against the spectral response curve $V(\lambda)$, which describes the relative sensitivity of the human eye to various wavelengths.[4] *Luminance* measures how bright a spectral power distribution appears to a human observer. For example, luminance accounts for the fact that an SPD with a particular amount of energy in the green wavelengths will appear much brighter to a human than an SPD with the same amount of energy in blue.

We will denote luminance here by Y; it related to spectral radiance $L(\lambda)$ by

$$Y = \int_\lambda L(\lambda)\, V(\lambda)\, d\lambda.$$

Luminance and the spectral response curve $V(\lambda)$ are closely related to the XYZ representation of color (Section 5.2.1). The CIE $Y(\lambda)$ tristimulus curve was chosen to be proportional to $V(\lambda)$ so that

$$Y = 683 \int_\lambda L(\lambda)\, Y(\lambda)\, d\lambda.$$

The units of luminance are candelas per meter squared (cd/m^2), where the candela is the photometric equivalent of radiant intensity. The quantity cd/m^2 is often referred to as a *nit*. Some representative luminance values are given in Table 5.1.

4 The spectral response curve model is based on experiments done in a normally illuminated indoor environment. Because sensitivity to color decreases in dark environments, it doesn't model the human visual system's response well under all lighting situations. Nonetheless, it forms the basis for the definition of luminance and other related photometric properties.

Table 5.1: Representative Luminance Values for a Number of Lighting Conditions.

Condition	Luminance (cd/m^2, or nits)
Sun at horizon	600,000
60-watt light bulb	120,000
Clear sky	8,000
Typical office	100–1000
Typical computer display	1–100
Street lighting	1–10
Cloudy moonlight	0.25

5.5 WORKING WITH RADIOMETRIC INTEGRALS

One of the most frequent tasks in rendering is the evaluation of integrals of radiometric quantities. In this section, we will present some tricks that can make this task easier. To illustrate the use of these techniques, we will use the computation of irradiance at a point as an example. Irradiance at a point p with surface normal $\mathbf{n}$ due to radiance over a set of directions Ω is

$$E(\mathrm{p}, \mathbf{n}) = \int_{\Omega} L_i(\mathrm{p}, \omega) \, |\cos\theta| \, \mathrm{d}\omega, \tag{5.3}$$

where $L_i(\mathrm{p}, \omega)$ is the incident radiance function (Figure 5.12) and the $\cos\theta$ term in this integral is due to the $\mathrm{d}A^{\perp}$ term in the definition of radiance. θ is measured as the angle between ω and the surface normal $\mathbf{n}$. Irradiance is usually computed over the hemisphere $\mathcal{H}^2(\mathbf{n})$ of directions about a given surface normal $\mathbf{n}$.

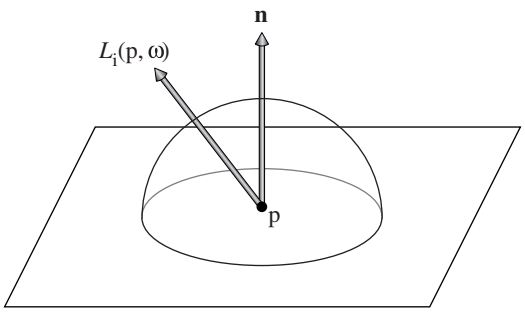


Figure 5.12: Irradiance at a point p is given by the integral of radiance times the cosine of the incident direction over the entire upper hemisphere above the point.

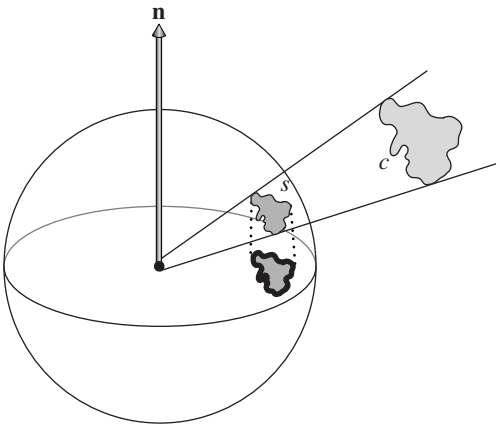


Figure 5.13: The projected solid angle subtended by an object c is the cosine-weighted solid angle that it subtends. It can be computed by finding the object's solid angle s, projecting it down to the plane perpendicular to the surface normal, and measuring its area there. Thus, the projected solid angle depends on the surface normal where it is being measured, since the normal orients the plane of projection.

5.5.1 INTEGRALS OVER PROJECTED SOLID ANGLE

The various cosine terms in the integrals for radiometric quantities can often distract from what is being expressed in the integral. This problem can be avoided using *projected solid angle* rather than solid angle to measure areas subtended by objects being integrated over. The projected solid angle subtended by an object is determined by projecting the object onto the unit sphere, as was done for the solid angle, but then projecting the resulting shape down onto the unit disk (Figure 5.13). Integrals over hemispheres of directions with respect to cosine-weighted solid angle can be rewritten as integrals over projected solid angle.

The projected solid angle measure is related to the solid angle measure by

$$\mathrm{d}\omega^{\perp} = |\cos\theta|\,\mathrm{d}\omega,$$

so the irradiance-from-radiance integral over the hemisphere can be written more simply as

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_{\mathrm{i}}(\mathrm{p}, \omega)\,\mathrm{d}\omega^{\perp}.$$

For the rest of this book, we will write integrals over directions in terms of solid angle, rather than projected solid angle. In other sources, however, projected solid angle can be common, and it is always important to be aware of the integrand's actual measure.

Just as we found irradiance in terms of incident radiance, we can also compute the total flux emitted from some object over the hemisphere surrounding the normal by

integrating over the object's surface area A:

$$\Phi = \int_A \int_{\mathcal{H}^2(\mathbf{n})} L_o(\mathrm{p}, \omega) \cos\theta \, \mathrm{d}\omega \, \mathrm{d}A$$
$$= \int_A \int_{\mathcal{H}^2(\mathbf{n})} L_o(\mathrm{p}, \omega) \, \mathrm{d}\omega^{\perp} \, \mathrm{d}A.$$

5.5.2 INTEGRALS OVER SPHERICAL COORDINATES

It is often convenient to transform integrals over solid angle into integrals over spherical coordinates (θ, ϕ). Recall that an (x, y, z) direction vector can also be written in terms of spherical angles (Figure 5.14):

$$x = \sin\theta \cos\phi$$
$$y = \sin\theta \sin\phi$$
$$z = \cos\theta.$$

In order to convert an integral over a solid angle to an integral over (θ, ϕ), we need to be able to express the relationship between the differential area of a set of directions $\mathrm{d}\omega$ and the differential area of a (θ, ϕ) pair (Figure 5.15). The differential area $\mathrm{d}\omega$ is the product of the differential lengths of its sides, $\sin\theta \, \mathrm{d}\phi$ and $\mathrm{d}\theta$. Therefore,

$$\mathrm{d}\omega = \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi. \quad [5.4]$$

We can thus see that the irradiance integral over the hemisphere, Equation (5.3) with

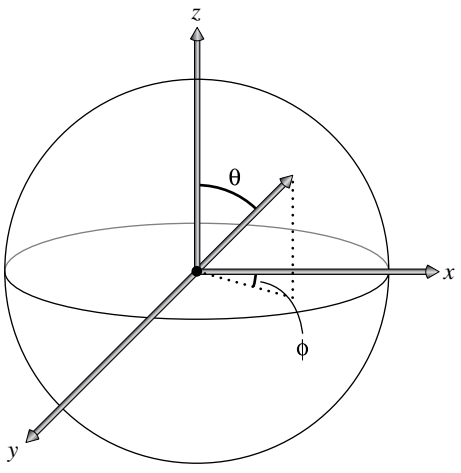


Figure 5.14: A direction vector can be written in terms of spherical coordinates (θ, ϕ) if the x, y, and z basis vectors are given as well. The spherical angle formulae make it easy to convert between the two representations.

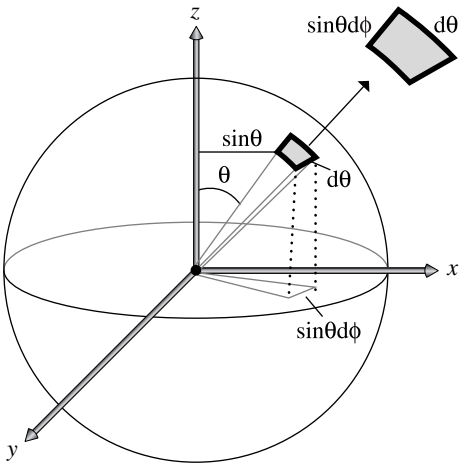


Figure 5.15: The differential area $\mathrm{d}A$ subtended by a differential solid angle is the product of the differential lengths of the two edges $\sin\theta\,\mathrm{d}\phi$ and $\mathrm{d}\theta$. The resulting relationship, $\mathrm{d}\omega = \sin\theta\,\mathrm{d}\theta\mathrm{d}\phi$, is the key to converting between integrals over solid angles and integrals over spherical angles.

$\Omega = \mathcal{H}^2(\mathbf{n})$, can equivalently be written as

$$E(\mathrm{p}, \mathbf{n}) = \int_0^{2\pi} \int_0^{\pi/2} L_\mathrm{i}(\mathrm{p}, \theta, \phi) \cos\theta \ \sin\theta \ \mathrm{d}\theta \, \mathrm{d}\phi$$

If the radiance is the same from all directions, this simplifies to $E = \pi L_\mathrm{i}$.

For convenience, we'll define two functions that convert θ and ϕ values into (x, y, z) direction vectors. The first function applies the earlier equations directly. Notice that these functions are passed the sine and cosine of θ, rather than θ itself. This is because the sine and cosine of θ are often already available to the caller. This is not normally the case for ϕ, however, so ϕ is passed in as is.

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector SphericalDirection(float sintheta,
                                 float costheta, float phi) {
    return Vector(sintheta * cosf(phi),
                  sintheta * sinf(phi),
                  costheta);
}
```

Vector 57

The second function takes three basis vectors representing the x, y, and z axes and returns the appropriate direction vector with respect to the coordinate frame defined by them:

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector SphericalDirection(float sintheta, float costheta,
                                 float phi, const Vector &x,
                                 const Vector &y, const Vector &z) {
    return sintheta * cosf(phi) * x +
           sintheta * sinf(phi) * y + costheta * z;
}
```

The conversion of a direction (x, y, z) to spherical angles can be found by

$$\theta = \arccos z$$
$$\phi = \arctan \frac{y}{x}.$$

The corresponding functions follow. Note that `SphericalTheta()` assumes that the vector `v` has been normalized before being passed in.

⟨*Geometry Inline Functions*⟩ +≡
```
inline float SphericalTheta(const Vector &v) {
    return acosf(Clamp(v.z, -1.f, 1.f));
}
```

⟨*Geometry Inline Functions*⟩ +≡
```
inline float SphericalPhi(const Vector &v) {
    float p = atan2f(v.y, v.x);
    return (p < 0.f) ? p + 2.f*M_PI : p;
}
```

5.5.3 INTEGRALS OVER AREA

One last transformation of integrals that can simplify computation is to turn integrals over directions into integrals over area. Consider the irradiance integral in Equation (5.3) again, and imagine there is a quadrilateral with constant outgoing radiance and we'd like to compute the resulting irradiance at a point p. Computing this value as an integral over directions is not straightforward, since given a particular direction it is nontrivial to determine if the quadrilateral is visible in that direction. It's much easier to compute the irradiance as an integral over the area of the quadrilateral.

Differential area is related to differential solid angle (as viewed from a point p) by

$$d\omega = \frac{dA \cos\theta}{r^2}, \tag{5.5}$$

where θ is the angle between the surface normal of dA and the vector to p, and r is the distance from p to dA (Figure 5.16). We will not derive this result here, but it can be understood intuitively: If dA is at distance 1 from p and is aligned exactly so that it is perpendicular to $d\omega$, then $d\omega = dA$, $\theta = 0$, and Equation (5.5) holds. As dA moves farther away from p, or as it rotates so that it's not aligned with the direction of $d\omega$, the r^2 and $\cos\theta$ terms compensate accordingly to reduce $d\omega$.

M_PI 1002
SphericalTheta() 292
Vector 57

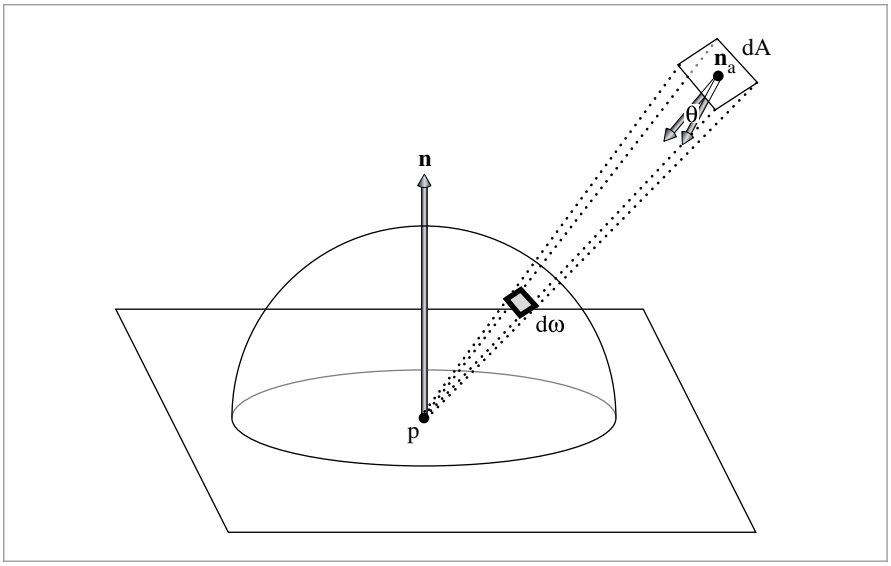


Figure 5.16: The differential solid angle subtended by a differential area $\mathrm{d}A$ is equal to $\mathrm{d}A \cos\theta / r^2$, where θ is the angle between $\mathrm{d}A$'s surface normal and the vector to the point p and r is the distance from p to $\mathrm{d}A$.

Therefore, we can write the irradiance integral for the quadrilateral source as

$$E(\mathrm{p}, \mathbf{n}) = \int_A L \cos\theta_\mathrm{i} \frac{\cos\theta_\mathrm{o}\, \mathrm{d}A}{r^2},$$

where L is the emitted radiance from the surface of the quadrilateral, θ_i is the angle between the surface normal at p and the direction from p to the point p′ on the light, and θ_o is the angle between the surface normal at p′ on the light and the direction from p′ to p (Figure 5.17).

5.6 SURFACE REFLECTION

When light is incident on a surface, the surface scatters the light, reflecting some of it back into the environment. There are two main effects that need to be described to model this reflection: the spectral distribution of the reflected light and its directional distribution. For example, the skin of a lemon mostly absorbs light in the blue wavelengths, but reflects most of the light in the red and green wavelengths (recall the lemon skin reflectance SPD in Figure 5.1). Therefore, when it is illuminated with white light, its color is yellow. The skin has pretty much the same color no matter what direction it's being observed from, although for some directions a highlight—a brighter area that is more white than yellow—is visible. In contrast, the light reflected from a point in a mirror depends almost entirely on the viewing direction. At a fixed point on the mirror, as the viewing angle changes, the object that is reflected in the mirror changes accordingly.

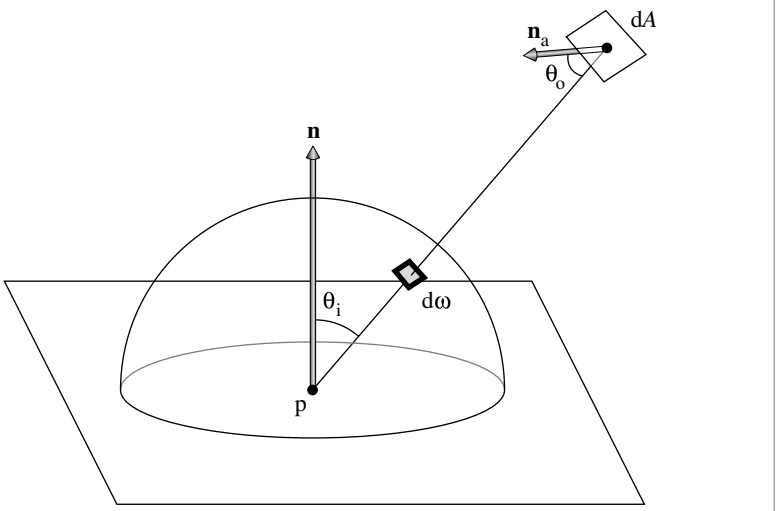


Figure 5.17: To compute irradiance at a point p from a quadrilateral source, it's easier to integrate over the surface area of the source than to integrate over the irregular set of directions that it subtends. The relationship between solid angles and areas given by Equation (5.5) lets us go back and forth between the two approaches.

Reflection from translucent surfaces is more complex; many materials ranging from skin and leaves to wax and dense liquids exhibit *subsurface light transport*, where light that enters the surface at one point exits it some distance away. (Consider, for example, how shining a flashlight in one's mouth makes one's cheeks light up, as light that enters the inside of the cheeks passes through the skin and exits the face.)

There are two abstractions for describing these mechanisms for light reflection: the BRDF and the BSSRDF, described in Sections 5.6.1 and 5.6.2, respectively. The BRDF describes surface reflection at a point neglecting the effect of subsurface light transport; for materials where this transport mechanism doesn't have a significant effect, this simplification introduces little error and makes the implementation of rendering algorithms much more efficient. The BSSRDF generalizes the BRDF and describes the more general setting of light reflection from translucent materials.

5.6.1 THE BRDF

The *bidirectional reflectance distribution function* (BRDF) gives a formalism for describing reflection from a surface. Consider the setting in Figure 5.18: we'd like to know how much radiance is leaving the surface in the direction ω_o toward the viewer, $L_o(p, \omega_o)$, as a result of incident radiance along the direction ω_i, $L_i(p, \omega_i)$.

If the direction ω_i is considered as a differential cone of directions, the differential irradiance at p is

$$dE(p, \omega_i) = L_i(p, \omega_i) \cos\theta_i \, d\omega_i. \quad [5.6]$$

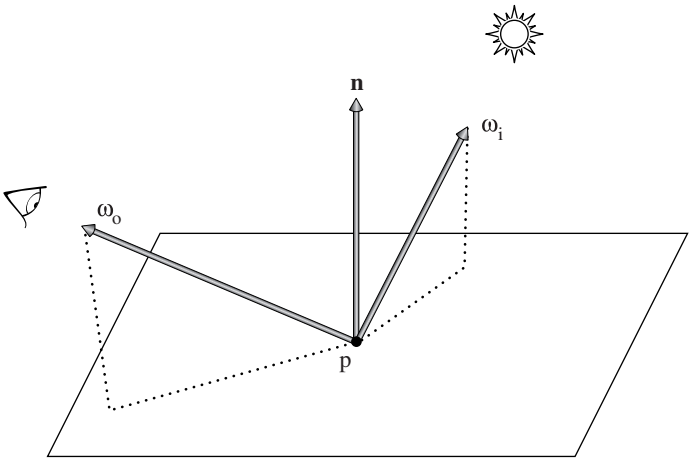


Figure 5.18: The BRDF. The bidirectional reflectance distribution function is a four-dimensional function over pairs of directions ω_i and ω_o that describes how much incident light along ω_i is scattered from the surface in the direction ω_o.

A differential amount of radiance will be reflected in the direction ω_o due to this irradiance. Because of the linearity assumption from geometric optics, the reflected differential radiance is proportional to the irradiance

$$dL_o(p, \omega_o) \propto dE(p, \omega_i).$$

The constant of proportionality defines the surface's BRDF for the particular pair of directions ω_i and ω_o:

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos\theta_i \, d\omega_i}. \quad [5.7]$$

Physically based BRDFs have two important qualities:

1. *Reciprocity:* For all pairs of directions ω_i and ω_o, $f_r(p, \omega_i, \omega_o) = f_r(p, \omega_o, \omega_i)$.
2. *Energy conservation:* The total energy of light reflected is less than or equal to the energy of incident light. For all directions ω_o,

$$\int_{\mathcal{H}^2(\mathbf{n})} f_r(p, \omega_o, \omega') \cos\theta' \, d\omega' \leq 1.$$

The surface's *bidirectional transmittance distribution function* (BTDF), which describes the distribution of transmitted light, can be defined in a manner similar to that for the BRDF. The BTDF is generally denoted by $f_t(p, \omega_o, \omega_i)$, where ω_i and ω_o are in opposite hemispheres around p. Remarkably, the BTDF does not obey reciprocity; we will discuss this issue in detail in Section 8.2.

For convenience in equations, we will denote the BRDF and BTDF when considered together as $f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})$; we will call this the *bidirectional scattering distribution function* (BSDF). Chapter 8 is entirely devoted to describing BSDFs that are used in graphics.

Using the definition of the BSDF, we have

$$\mathrm{d}L_\mathrm{o}(\mathrm{p}, \omega_\mathrm{o}) = f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}.$$

Here an absolute value has been added to the $\cos\theta_\mathrm{i}$ term. This is done because surface normals in `pbrt` are not reoriented to lie on the same side of the surface as ω_i (many other rendering systems do this, although we find it more useful to leave them in their natural orientation as given by the `Shape`). Doing so makes it easier to consistently apply conventions like "the surface normal is assumed to point outside the surface" elsewhere in the system. Thus, applying the absolute value to $\cos\theta$ terms like these ensures that the desired quantity is actually calculated.

We can integrate this equation over the sphere of incident directions around p to compute the outgoing radiance in direction ω_o due to the incident illumination at p from all directions:

$$L_\mathrm{o}(\mathrm{p}, \omega_\mathrm{o}) = \int_{\mathbb{S}^2} f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}. \tag{5.8}$$

This is a fundamental equation in rendering; it describes how an incident distribution of light at a point is transformed into an outgoing distribution, based on the scattering properties of the surface. It is often called the *scattering equation* when the sphere $\mathbb{S}^2$ is the domain (as it is here), or the *reflection equation* when just the upper hemisphere $\mathcal{H}^2(\mathbf{n})$ is being integrated over. One of the key tasks of the integration routines in Chapter 15 is to evaluate this integral at points on surfaces in the scene.

5.6.2 THE BSSRDF

Many materials exhibit a significant amount of subsurface light transport: light that enters them at one point may exit quite far away. In general, subsurface light transport is one of the key mechanisms that gives objects like wax, candles, marble, skin, and many biological tissues their characteristic appearances.

The *bidirectional scattering-surface reflectance distribution function* (BSSRDF) is the formalism that describes these kinds of scattering processes. It is a distribution function $S(\mathrm{p_o}, \omega_\mathrm{o}, \mathrm{p_i}, \omega_\mathrm{i})$ that describes the ratio of exitant differential radiance at point $\mathrm{p_o}$ in direction ω_o to the differential irradiance at $\mathrm{p_i}$ from direction ω_i:

$$S(\mathrm{p_o}, \omega_\mathrm{o}, \mathrm{p_i}, \omega_\mathrm{i}) = \frac{\mathrm{d}L_\mathrm{o}(\mathrm{p_o}, \omega_\mathrm{o})}{\mathrm{d}E(\mathrm{p_i}, \omega_\mathrm{i})}.$$

The generalization of the scattering equation for the BSSRDF requires integration over surface area *and* incoming direction, turning the 2D scattering Equation (5.8) into a 4D integral. With two more dimensions to integrate over, it is substantially more complex to use in rendering algorithms.

$$L_\mathrm{o}(\mathrm{p_o}, \omega_\mathrm{o}) = \int_A \int_{\mathcal{H}^2(\mathbf{n})} S(\mathrm{p_o}, \omega_\mathrm{o}, \mathrm{p_i}, \omega_\mathrm{i})\, L_\mathrm{i}(\mathrm{p_i}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}\, \mathrm{d}A. \tag{5.9}$$

Shape 108

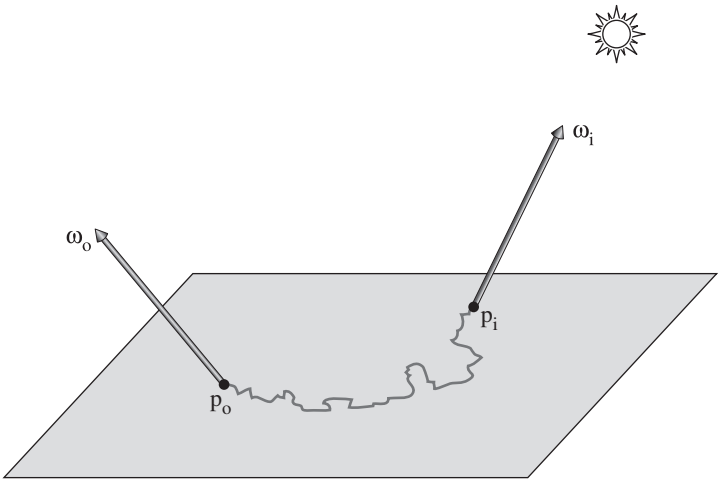


Figure 5.19: The bidirectional scattering-surface reflectance distribution function generalizes the BSDF to account for light that exits the surface at a point other than where it enters. It is often more difficult to evaluate than the BSDF, although subsurface light transport can make a substantial contribution to the appearance of many real-world objects.

However, as the distance between points p_i and p_o increases, the value of S generally diminishes. This fact can be a substantial help in implementations of subsurface scattering algorithms.

Light transport beneath a surface is described by the same principles as volume light transport in participating media and is described by the equation of transfer, which is introduced in Section 16.1. Subsurface scattering is thus based on the same effects as light scattering in clouds and smoke, just at a smaller scale. The `DipoleSubsurfaceIntegrator` in Section 16.5 implements an algorithm that applies an approximation to efficiently approximate this integral.

FURTHER READING

Meyer was one of the first researchers to closely investigate spectral representations in graphics (Meyer and Greenberg 1980; Meyer et al. 1986). Hall (1989) summarized the state of the art in spectral representations through 1989, and Glassner's *Principles of Digital Image Synthesis* (1995) covers the topic through the mid-1990s. Survey articles by Hall (1999), Johnson and Fairchild (1999), and Delvin et al. (2002) are good resources on this topic.

`DipoleSubsurfaceIntegrator` 887

Borges (1991) analyzed the error introduced from the tristimulus representation when used for spectral computation. A polynomial representation for spectra was proposed by Raso and Fournier (1991). Peercy (1993) developed a technique based on choosing

basis functions in a scene-dependent manner: by looking at the SPDs of the lights and reflecting objects in the scene, a small number of basis functions that could accurately represent the scene's SPDs were found using characteristic vector analysis. Rougeron and Péroche (1997) projected all SPDs in the scene onto a hierarchical basis (the Haar wavelets), showing that this adaptive representation can be used to stay within a desired error bound. Ward and Eydelberg-Vileshin (2002) developed a method for improving the spectral results from a regular RGB-only rendering system by carefully adjusting the color values provided to the system before rendering.

Another approach to spectral representation was investigated by Sun et al. (2001), who partitioned SPDs into a smooth base SPD and a set of spikes. Each part was represented differently, using basis functions that worked well for each of these parts of the distribution. Drew and Finlayson (2003) applied a "sharp" basis, which is adaptive but has the property that computing the product of two functions in the basis doesn't require a full matrix multiplication as many other basis representations do.

Evans and McCool (1999) introduced stratified wavelength clusters for representing SPDs: the idea is that each spectral computation uses a small fixed number of samples at representative wavelengths, chosen according to the spectral distribution of the light source. Subsequent computations use different wavelengths, such that individual computations are relatively efficient (being based on a small number of samples), but, in the aggregate over a large number of computations, the overall range of wavelengths is well covered. Related to this approach is the idea of computing the result for just a single wavelength in each computation and averaging the results together: this was the method used by Walter et al. (1997) and Morley et al. (2006), who showed that this approach converges reasonably quickly.

Glassner (1989b) has written an article on the underconstrained problem of converting RGB values (e.g., as selected by the user from a display) to an SPD. Smits (1999) developed an improved method that is the one we have implemented in Section 5.2.2.

McCluney's book on radiometry is an excellent introduction to the topic (McCluney 1994). Preisendorfer (1965) also covered radiometry in an accessible manner and delved into the relationship between radiometry and the physics of light. Nicodemus et al. (1977) carefully defined the BRDF, BSSRDF, and various quantities that can be derived from them. See Moon and Spencer (1936, 1948) and Gershun (1939) for classic early introductions to radiometry. Finally, Lambert's seminal early writings about photometry from the mid-18th century have been translated into English by DiLaura (Lambert 1760).

EXERCISES

2 5.1 Implement a new spectral basis functions representation in `pbrt`. Compare both image quality and rendering time to the `RGBSpectrum` and `SampledSpectrum` representations implemented in this chapter. Be sure to include tricky situations like fluorescent lighting.

5.2 Compute the irradiance at a point due to a unit-radius disk h units directly above its normal with constant outgoing radiance of 10 J/m^2 sr. Do the computation twice, once as an integral over solid angle and once as an integral over area. (Hint: If the results don't match at first, see Section 13.6.2.)

5.3 Similarly, compute the irradiance at a point due to a square quadrilateral with outgoing radiance of 10 J/m^2 sr that has sides of length 1 and is 1 unit directly above the point in the direction of its surface normal.

CHAPTER SIX



06 CAMERA MODELS

In Chapter 1, we described the pinhole camera model that is commonly used in computer graphics. This model is easy to describe and simulate, but it has a number of drawbacks. For example, everything rendered with a pinhole camera is in sharp focus, which can make images look computer generated. In order to make images that appear realistic, it is important to better simulate properties of real-world imaging systems.

Like the `Shapes` from Chapter 3, cameras in `pbrt` are represented by an abstract base class. This chapter describes the `Camera` class and its two closely related methods: `Camera::GenerateRay()` and `Camera::GenerateRayDifferential()`. The first method computes the world space ray corresponding to a sample position on the image plane. By generating these rays in different ways based on different models of image formation, the cameras in `pbrt` can create many types of images of the same 3D scene. The second method not only generates this ray but also computes information about the image area that the ray is sampling; this information is used for anti-aliasing computations in Chapter 10, for example. Here, we will show a few implementations of the `Camera` interface, each of which generates rays in a different way to simulate different imaging processes.

6.1 CAMERA MODEL

The abstract `Camera` base class holds generic camera options and defines the interface that all camera implementations must provide. It is defined in the files `core/camera.h` and `core/camera.cpp`.

⟨*Camera Declarations*⟩ ≡
```
class Camera {
public:
    ⟨Camera Interface 302⟩
    ⟨Camera Public Data 302⟩
};
```

The base `Camera` constructor takes several parameters that are appropriate for all camera types. One of the most important is the transformation that places the camera in the scene, which is stored in the `CameraToWorld` member variable. The `Camera` stores an `AnimatedTransform` to place itself in the scene (rather than just a regular `Transform`) so that the camera itself can be moving over time.

Real-world cameras have a shutter that opens for a short period of time to expose the film to light. One result of this nonzero exposure time is *motion blur*: objects that move during the film exposure time are blurred. If time values between the shutter open time and the shutter close time are associated with each ray, it is possible to compute images that exhibit motion blur. All `Camera`s therefore store a shutter open and shutter close time and are responsible for generating rays with appropriately set times. `Camera`s also contain an instance of the `Film` class to represent the final image to be computed. `Film` will be described in Section 7.8.

Camera implementations must pass along parameters that set these values to the `Camera` constructor. We will only show its prototype here because its implementation just copies the parameters to the corresponding member variables.

⟨*Camera Interface*⟩ ≡ 302
```
Camera(const AnimatedTransform &cam2world, float sopen, float sclose,
       Film *film);
```

⟨*Camera Public Data*⟩ ≡ 302
```
AnimatedTransform CameraToWorld;
const float shutterOpen, shutterClose;
Film *film;
```

The first method that camera subclasses need to implement is `Camera::GenerateRay()`, which generates a ray for a given image sample. The `CameraSample::imageX` and `CameraSample::imageY` components of the camera sample give the raster space x and y coordinates on the image plane for the ray (the complete contents of the `CameraSample` structure are described in detail in Chapter 7). It is important that the camera normalize the direction component of the returned ray—many other parts of the system will depend on this behavior.

This method also returns a floating-point value that gives a weight for how much light arriving at the film plane along the generated ray will contribute to the final image. Most cameras always return a value of one, but cameras that simulate real physical lens systems might need to set this value to indicate how much light the ray carries through the lenses based on their optical properties.

AnimatedTransform 96
Camera 302
Camera::GenerateRay() 303
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
Film 403
Transform 76

⟨*Camera Interface*⟩ +≡ 302
```
virtual float GenerateRay(const CameraSample &sample,
                          Ray *ray) const = 0;
```

The `GenerateRayDifferential()` method computes a main ray like `GenerateRay()` but also computes the corresponding rays for pixels shifted one pixel in the x and y directions on the image plane. This information helps give the rest of the system a notion of how much of the image area a particular camera ray's sample represents. Many `Camera` implementations will be able to compute these rays directly, but here we provide a default implementation that initializes shifted `CameraSample`s and calls the camera's `GenerateRay()` method repeatedly.

⟨*Camera Method Definitions*⟩ ≡
```
float Camera::GenerateRayDifferential(const CameraSample &sample,
                                      RayDifferential *rd) const {
    float wt = GenerateRay(sample, rd);
    ⟨Find ray after shifting one pixel in the x direction 303⟩
    ⟨Find ray after shifting one pixel in the y direction⟩
    if (wtx == 0.f || wty == 0.f) return 0.f;
    rd->hasDifferentials = true;
    return wt;
}
```

Finding the ray for one pixel over in x is just a matter of initializing a new `CameraSample` and copying the appropriate values into the `RayDifferential` structure. The implementation of the fragment ⟨*Find ray after shifting one pixel in the y direction*⟩ follows similarly and isn't included here.

⟨*Find ray after shifting one pixel in the x direction*⟩ ≡ 303
```
CameraSample sshift = sample;
++(sshift.imageX);
Ray rx;
float wtx = GenerateRay(sshift, &rx);
rd->rxOrigin = rx.o;
rd->rxDirection = rx.d;
```

Camera::GenerateRay() 303
CameraSample 342
CameraSample::imageX 342
Ray 66
Ray::d 67
Ray::o 67
RayDifferential 69
RayDifferential::
hasDifferentials 69
RayDifferential::
rxDirection 69
RayDifferential::rxOrigin 69

6.1.1 CAMERA COORDINATE SPACES

We have already made use of two important modeling coordinate spaces, object space and world space. We will now introduce four useful coordinate spaces related to the camera and imaging, summarized in Figure 6.1. All together, we now have the following:

- *Object space:* This is the coordinate system in which geometric primitives are defined. For example, spheres in `pbrt` are defined to be centered at the origin of object space.
- *World space:* While each primitive may have its own object space, all objects in the scene are placed in relation to a single world space. Each primitive has an object-to-world transformation that determines where it is located in world space. World space is the standard frame that all spaces are defined in terms of.

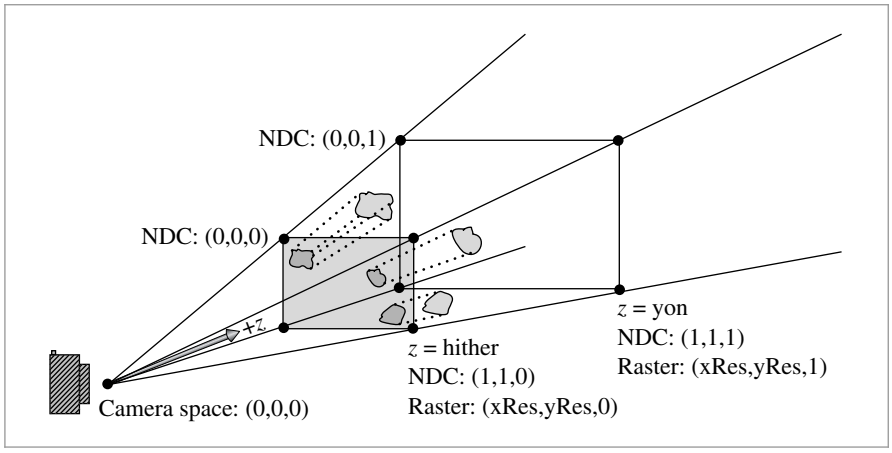


Figure 6.1: A handful of camera-related coordinate spaces help to simplify the implementation of `Camera`s. The camera class holds transformations between them. Scene objects in world space are viewed by the camera, which sits at the origin of camera space and points along the $+z$ axis. Objects between the hither and yon planes are projected onto the image plane at $z = \text{hither}$ in camera space. The image plane is at $z = 0$ in raster space, where x and y range from $(0, 0)$ to $(\text{xResolution}, \text{yResolution})$. Normalized device coordinate (NDC) space normalizes raster space so that x and y range from $(0, 0)$ to $(1, 1)$.

- *Camera space:* A virtual camera is placed in the scene at some world space point with a particular viewing direction and orientation. This defines a new coordinate system with its origin at the camera's location. The z axis of this coordinate system is mapped to the viewing direction, and the y axis is mapped to the up direction. This is a handy space for reasoning about which objects are potentially visible to the camera. For example, if an object's camera space bounding box is entirely behind the $z = 0$ plane (and the camera doesn't have a field of view wider than 180 degrees), the object will not be visible to the camera.
- *Screen space:* Screen space is defined on the image plane. The camera projects objects in camera space onto the image plane; the parts inside the *screen window* are visible in the image that is generated. Depth z values in screen space range from zero to one, corresponding to points at the near and far clipping planes, respectively. Note that, although this is called "screen" space, it is still a 3D coordinate system, since z values are meaningful.
- *Normalized device coordinate (NDC) space:* This is the coordinate system for the actual image being rendered. In x and y, this space ranges from $(0, 0)$ to $(1, 1)$, with $(0, 0)$ being the upper-left corner of the image. Depth values are the same as in screen space and a linear transformation converts from screen to NDC space.
- *Raster space:* This is almost the same as NDC space, except the x and y coordinates range from $(0, 0)$ to (xResolution, yResolution).

All cameras store a world-space-to-camera-space transformation; this can be used to transform primitives in the scene into camera space. The origin of camera space is the camera's position, and the camera points along the camera space z axis. The projective cameras in the next section use 4×4 matrices to transform between all of these spaces

Camera 302

as needed, but cameras with unusual imaging characteristics can't necessarily represent all of these transformations with matrices.

6.2 PROJECTIVE CAMERA MODELS

One of the fundamental issues in 3D computer graphics is the *3D viewing problem:* how to project a three-dimensional scene onto a two-dimensional image for display. Most of the classic approaches can be expressed by a 4×4 projective transformation matrix. Therefore, we will introduce a projection matrix camera class and then define two camera models based on it. The first implements an orthographic projection, and the other implements a perspective projection—two classic and widely used projections.

⟨*Camera Declarations*⟩ +≡
```
class ProjectiveCamera : public Camera {
public:
    ⟨ProjectiveCamera Public Methods⟩
protected:
    ⟨ProjectiveCamera Protected Data 305⟩
};
```

In addition to the parameters required by the `Camera` base class, the `ProjectiveCamera` takes the projective transformation matrix, screen space extent of the image, and additional parameters for depth of field. Depth of field, which will be described and implemented at the end of this section, simulates the blurriness of out-of-focus objects that occurs in real lens systems.

⟨*Camera Method Definitions*⟩ +≡
```
ProjectiveCamera::ProjectiveCamera(const AnimatedTransform &cam2world,
        const Transform &proj, const float screenWindow[4], float sopen,
        float sclose, float lensr, float focald, Film *f)
    : Camera(cam2world, sopen, sclose, f) {
    ⟨Initialize depth of field parameters 314⟩
    ⟨Compute projective camera transformations 305⟩
}
```

`ProjectiveCamera` implementations pass the projective transformation up to the base class constructor shown here. This transformation gives the camera-to-screen projection; from that, the constructor can compute most of the others that are needed.

⟨*Compute projective camera transformations*⟩ ≡ 305
```
CameraToScreen = proj;
⟨Compute projective camera screen transformations 306⟩
RasterToCamera = Inverse(CameraToScreen) * RasterToScreen;
```

⟨*ProjectiveCamera Protected Data*⟩ ≡ 305
```
Transform CameraToScreen, RasterToCamera;
```

The only nontrivial transformation to compute in the constructor is the screen-to-raster projection. In the following code, note the composition of transformations where (reading from bottom to top), we start with a point in screen space, translate so that the

AnimatedTransform 96
Camera 302
Film 403
ProjectiveCamera 305
ProjectiveCamera::
CameraToScreen 305
ProjectiveCamera::
RasterToCamera 305
Transform 76

upper-left corner of the screen is at the origin, and then scale by the reciprocal of the screen width and height, giving us a point with x and y coordinates between zero and one (these are NDC coordinates). Finally, we scale by the raster resolution, so that we end up covering the entire raster range from (0, 0) up to the overall raster resolution. An important detail here is that the y coordinate is inverted by this transformation; this is necessary because increasing y values move up the image in screen coordinates, but down in raster coordinates.

⟨*Compute projective camera screen transformations*⟩ ≡ 305
```
ScreenToRaster = Scale(float(film->xResolution),
                       float(film->yResolution), 1.f) *
    Scale(1.f / (screenWindow[1] - screenWindow[0]),
          1.f / (screenWindow[2] - screenWindow[3]), 1.f) *
    Translate(Vector(-screenWindow[0], -screenWindow[3], 0.f));
RasterToScreen = Inverse(ScreenToRaster);
```

⟨*ProjectiveCamera Protected Data*⟩ +≡ 305
```
Transform ScreenToRaster, RasterToScreen;
```

6.2.1 ORTHOGRAPHIC CAMERA

⟨*OrthographicCamera Declarations*⟩ ≡
```
class OrthoCamera : public ProjectiveCamera {
public:
    ⟨OrthoCamera Public Methods⟩
private:
    ⟨OrthoCamera Private Data 307⟩
};
```

The orthographic camera, defined in the files `cameras/orthographic.h` and `cameras/orthographic.cpp`, is based on the orthographic projection transformation. The orthographic transformation takes a rectangular region of the scene and projects it onto the front face of the box that defines the region. It doesn't give the effect of *foreshortening*—objects becoming smaller on the image plane as they get farther away—but it does leave parallel lines parallel, and it preserves relative distance between objects. Figure 6.2 shows how this rectangular volume defines the visible region of the scene. Figure 6.3 compares the result of using the orthographic projection for rendering to the perspective projection defined in the next section.

The orthographic camera constructor generates the orthographic transformation matrix with the `Orthographic()` function, which will be defined shortly.

⟨*OrthographicCamera Definitions*⟩ ≡
```
OrthoCamera::OrthoCamera(const AnimatedTransform &cam2world,
        const float screenWindow[4], float sopen, float sclose,
        float lensr, float focald, Film *f)
    : ProjectiveCamera(cam2world, Orthographic(0., 1.), screenWindow,
                       sopen, sclose, lensr, focald, f) {
    ⟨Compute differential changes in origin for ortho camera rays 307⟩
}
```

AnimatedTransform 96
Film 403
Film::xResolution 403
Film::yResolution 403
OrthoCamera 306
Orthographic() 307
ProjectiveCamera 305
ProjectiveCamera::
RasterToScreen 306
ProjectiveCamera::
ScreenToRaster 306
Scale() 80
Transform 76
Translate() 79
Vector 57

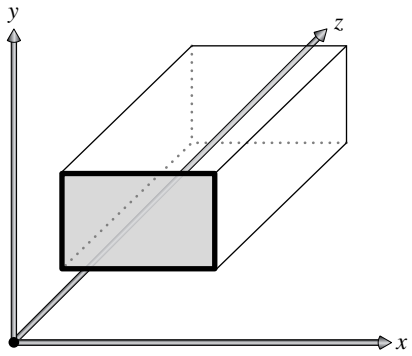


Figure 6.2: The orthographic view volume is an axis-aligned box in camera space, defined such that objects inside the region are projected onto the $z =$ hither face of the box.

The orthographic viewing transformation leaves x and y coordinates unchanged, but maps z values at the hither plane to 0 and z values at the yon plane to 1. To do this, the scene is first translated along the z axis so that the hither plane is aligned with $z = 0$. Then, the scene is scaled in z so that the yon plane maps to $z = 1$. The composition of these two transformations gives the overall transformation. (For a ray tracer like `pbrt`, we'd like the hither plane to be at 0 so that rays start at the plane that goes through the camera's position; the yon plane offset doesn't particularly matter.)

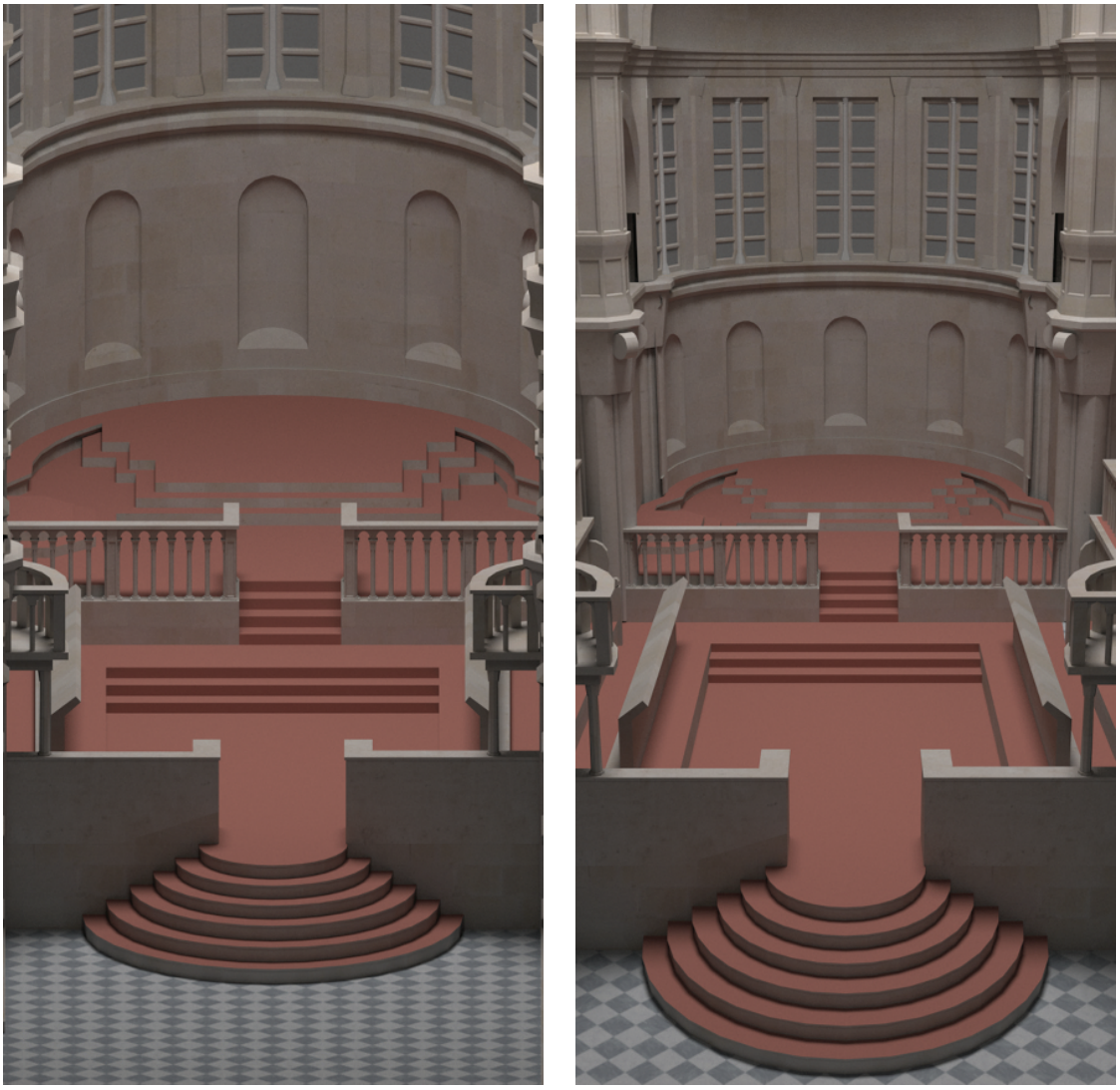
⟨*Transform Method Definitions*⟩ +≡
```
Transform Orthographic(float znear, float zfar) {
    return Scale(1.f, 1.f, 1.f / (zfar-znear)) *
           Translate(Vector(0.f, 0.f, -znear));
}
```

Thanks to the simplicity of the orthographic projection, it's easy to directly compute the differential rays in the x and y directions in the `GenerateRayDifferential()` method. The directions of the differential rays will be the same as the main ray (as they are for all rays generated by an orthographic camera), and the difference in origins will be the same for all rays. Therefore, the constructor here precomputes how much the ray origins shift in camera space coordinates due to a single pixel shift in the x and y directions on the image plane.

ProjectiveCamera::
RasterToCamera 305
Scale() 80
Transform 76
Translate() 79
Vector 57

⟨*Compute differential changes in origin for ortho camera rays*⟩ ≡ 306
```
dxCamera = RasterToCamera(Vector(1, 0, 0));
dyCamera = RasterToCamera(Vector(0, 1, 0));
```

⟨*OrthoCamera Private Data*⟩ ≡ 306
```
Vector dxCamera, dyCamera;
```



(a) (b)

Figure 6.3: Images of the Church Model. Rendered with (a) orthographic and (b) perspective cameras. Note that features like the stairs, checks on the floor, and back windows are rendered quite differently with the two models. The lack of foreshortening makes the orthographic view feel like it has less depth, although it does preserve parallel lines, which can be a useful property.

We can now go through the code to take a sample point in raster space and turn it into a camera ray. The process is summarized in Figure 6.4. First, the raster space sample position is transformed into a point in camera space, giving a point located on the hither plane, which is the origin of the camera ray. Because the camera space viewing direction points down the z axis, the camera space ray direction is (0, 0, 1).

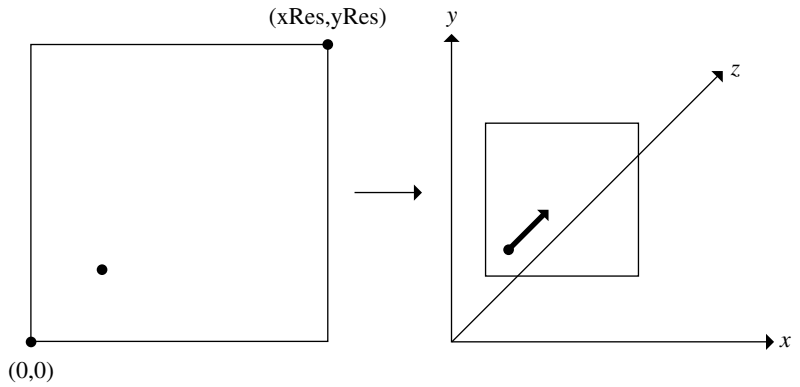


Figure 6.4: To create a ray with the orthographic camera, a raster space position on the image plane is transformed to camera space, giving the ray's origin on the hither plane. The ray's direction in camera space is $(0, 0, 1)$, down the z axis.

If depth of field has been enabled for this scene, the ray's origin and direction are modified so that depth of field is simulated. Depth of field will be explained later in this section. The ray's time value is set by linearly interpolating between the shutter open and shutter close times by the `CameraSample::time` offset (which is in the range $[0, 1)$). Finally, the ray is transformed into world space before being returned.

⟨*OrthographicCamera Definitions*⟩ +≡
```
float OrthoCamera::GenerateRay(const CameraSample &sample, Ray *ray) const {
    ⟨Generate raster and camera samples 309⟩
    *ray = Ray(Pcamera, Vector(0,0,1), 0.f, INFINITY);
    ⟨Modify ray for depth of field 315⟩
    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

Camera::CameraToWorld 302
Camera::shutterClose 302
Camera::shutterOpen 302
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::time 342
INFINITY 1002
Lerp() 1000
OrthoCamera 306
Point 63
ProjectiveCamera::
RasterToCamera 305
Ray 66
Vector 57

Once all of the transformation matrices have been set up, it's easy to set up the raster space sample point and transform it to camera space.

⟨*Generate raster and camera samples*⟩ ≡ 309, 312
```
Point Pras(sample.imageX, sample.imageY, 0);
Point Pcamera;
RasterToCamera(Pras, &Pcamera);
```

The implementation of `GenerateRayDifferential()` performs the same computation to generate the main camera ray. The differential ray origins are found using the offsets computed in the `OrthoCamera` constructor and then the full ray differential is transformed to world space.

⟨*OrthographicCamera Definitions*⟩ +≡
```
float OrthoCamera::GenerateRayDifferential(const CameraSample &sample,
        RayDifferential *ray) const {
    ⟨Compute main orthographic viewing ray⟩
    ray->rxOrigin = ray->o + dxCamera;
    ray->ryOrigin = ray->o + dyCamera;
    ray->rxDirection = ray->ryDirection = ray->d;
    ray->hasDifferentials = true;
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

6.2.2 PERSPECTIVE CAMERA

The perspective projection is similar to the orthographic projection in that it projects a volume of space onto a 2D image plane. However, it includes the effect of foreshortening: objects that are far away are projected to be smaller than objects of the same size that are closer. Unlike the orthographic projection, the perspective projection doesn't preserve distances or angles, and parallel lines no longer remain parallel. The perspective projection is a reasonably close match to how an eye or camera lens generates images of the three-dimensional world. The perspective camera is implemented in the files `cameras/perspective.h` and `cameras/perspective.cpp`.

⟨*PerspectiveCamera Declarations*⟩ ≡
```
class PerspectiveCamera : public ProjectiveCamera {
public:
    ⟨PerspectiveCamera Public Methods 312⟩
private:
    ⟨PerspectiveCamera Private Data 312⟩
};
```

⟨*PerspectiveCamera Method Definitions*⟩ ≡
```
PerspectiveCamera:: PerspectiveCamera(const AnimatedTransform &cam2world,
        const float screenWindow[4], float sopen, float sclose,
        float lensr, float focald, float fov, Film *f)
    : ProjectiveCamera(cam2world, Perspective(fov, 1e-2f, 1000.f),
                       screenWindow, sopen, sclose, lensr, focald, f) {
    ⟨Compute differential changes in origin for perspective camera rays 312⟩
}
```

The perspective projection describes perspective viewing of the scene. Points in the scene are projected onto a viewing plane perpendicular to the z axis. The `Perspective()` function computes this transformation; it takes a field-of-view angle in `fov`, and the distances to a near z plane and a far z plane. After the perspective projection, points at the near z plane are mapped to have $z = 0$ and points at the far plane have $z = 1$. (Figure 6.5). For rendering systems based on rasterization, it's important to set the positions of these planes carefully; they determine the z range of the scene that is rendered, but setting them with too many orders of magnitude variation between their values can lead to numeric precision errors. For a ray tracers like `pbrt`, they can be set arbitrarily as they are here.

AnimatedTransform 96
Camera::CameraToWorld 302
CameraSample 342
Film 403
OrthoCamera 306
OrthoCamera::dxCamera 307
OrthoCamera::dyCamera 307
Perspective() 311
PerspectiveCamera 310
ProjectiveCamera 305
Ray::d 67
Ray::o 67
RayDifferential 69
RayDifferential::
hasDifferentials 69
RayDifferential::
rxDirection 69
RayDifferential::rxOrigin 69
RayDifferential::
ryDirection 69
RayDifferential::ryOrigin 69

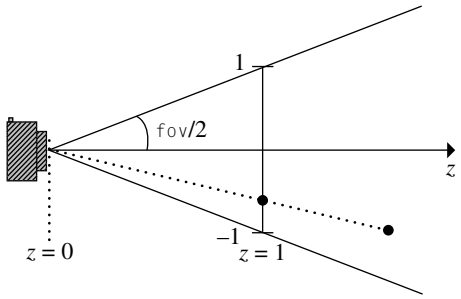


Figure 6.5: The perspective transformation matrix projects points in camera space onto the image plane. The x' and y' coordinates of the projected points are equal to the unprojected x and y coordinates divided by the z coordinate. The projected z' coordinate is computed so that points on the near plane map to $z' = 0$ and points on the far plane map to $z' = 1$.

⟨*Transform Method Definitions*⟩ +≡
```
Transform Perspective(float fov, float n, float f) {
    ⟨Perform projective divide 311⟩
    ⟨Scale to canonical viewing volume 312⟩
}
```

The transformation is most easily understood in two steps:

1. Points p in camera space are projected onto the viewing plane. A bit of algebra shows that the projected x' and y' coordinates on the viewing plane can be computed by dividing x and y by the point's z coordinate value. The projected z depth is remapped so that z values at the hither plane are 0 and z values at the yon plane are 1. The computation we'd like to do is

$$x' = x/z$$
$$y' = y/z$$
$$z' = \frac{f(z-n)}{z(f-n)}.$$

 All of this computation can be encoded in a 4×4 matrix using homogeneous coordinates:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

⟨*Perform projective divide*⟩ ≡ 311
```
Matrix4x4 persp = Matrix4x4(1, 0,          0,              0,
                            0, 1,          0,              0,
                            0, 0, f / (f - n), -f*n / (f - n),
                            0, 0,          1,              0);
```
Matrix4x4 1021
Transform 76

2. The angular field of view (`fov`) specified by the user is accounted for by scaling the (x, y) values on the projection plane so that points inside the field of view project to coordinates between $[-1, 1]$ on the view plane. For square images, both x and y lie between $[-1, 1]$ in screen space. Otherwise, the direction in which the image is narrower maps to $[-1, 1]$ and the wider direction maps to a proportionally larger range of screen space values. Recall that the tangent is equal to the ratio of the opposite side of a right triangle to the adjacent side. Here the adjacent side has length 1, so the opposite side has the length $\tan(\texttt{fov}/2)$. Scaling by the reciprocal of this length maps the field of view to range from $[-1, 1]$.

⟨*Scale to canonical viewing volume*⟩ ≡ 311
```
float invTanAng = 1.f / tanf(Radians(fov) / 2.f);
return Scale(invTanAng, invTanAng, 1) * Transform(persp);
```

Similar to the `OrthoCamera`, information about how the camera rays generated by the `PerspectiveCamera` change as we shift pixels on the image plane can be precomputed in the constructor. Here, we compute the change in position on the near perspective plane in camera space with respect to shifts in pixel location.

⟨*Compute differential changes in origin for perspective camera rays*⟩ ≡ 310
```
dxCamera = RasterToCamera(Point(1,0,0)) - RasterToCamera(Point(0,0,0));
dyCamera = RasterToCamera(Point(0,1,0)) - RasterToCamera(Point(0,0,0));
```

⟨*PerspectiveCamera Private Data*⟩ ≡ 310
```
Vector dxCamera, dyCamera;
```

With the perspective projection, all rays originate from the origin, (0, 0, 0), in camera space. A ray's direction is given by the vector from the origin to the point on the near plane `Pcamera` that corresponds to the provided `CameraSample`. In other words, the ray's vector direction is componentwise equal to this point's position, so rather than doing a useless subtraction to compute the direction, we just initialize the direction directly from the point `Pcamera`.

⟨*PerspectiveCamera Method Definitions*⟩ +≡
```
float PerspectiveCamera::GenerateRay(const CameraSample &sample,
                                     Ray *ray) const {
    ⟨Generate raster and camera samples 309⟩
    *ray = Ray(Point(0,0,0), Vector(Pcamera), 0.f, INFINITY);
    ⟨Modify ray for depth of field 315⟩
    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

The `GenerateRayDifferential()` method follows the implementation of `GenerateRay()`, except for an additional fragment that computes the differential rays.

⟨*PerspectiveCamera Public Methods*⟩ ≡ 310
```
float GenerateRayDifferential(const CameraSample &sample,
                              RayDifferential *ray) const;
```

Camera::CameraToWorld 302
Camera::shutterClose 302
Camera::shutterOpen 302
CameraSample 342
CameraSample::time 342
INFINITY 1002
Lerp() 1000
OrthoCamera 306
PerspectiveCamera 310
PerspectiveCamera::dxCamera 312
PerspectiveCamera::dyCamera 312
Point 63
ProjectiveCamera::RasterToCamera 305
Radians() 1001
Ray 66
Ray::time 67
RayDifferential 69
Scale() 80
Transform 76
Vector 57

⟨*Compute offset rays for* PerspectiveCamera *ray differentials*⟩ ≡

```
ray->rxOrigin = ray->ryOrigin = ray->o;
ray->rxDirection = Normalize(Vector(Pcamera) + dxCamera);
ray->ryDirection = Normalize(Vector(Pcamera) + dyCamera);
```

6.2.3 DEPTH OF FIELD

Real cameras have lens systems that focus light through a finite-sized aperture onto the film plane. Because the aperture has finite area, a single point in the scene may be projected onto an area on the film plane called the *circle of confusion* (Figure 6.6). Correspondingly, a finite area of the scene may be visible from a single point on the image plane, giving a blurred image. Figures 6.7 and 6.8 show this effect, depth of field, in a scene with a series of copies of the dragon model. Figure 6.7(a) is rendered with an infinitesimal aperture and thus without any depth of field effects. Figures 6.7(b) and 6.8 show the increase in blurriness as the size of the lens aperture is increased. Note that the second dragon from the right remains in focus throughout all of the images, as the plane of focus has been placed at its depth. Figure 6.9 shows depth of field used to render the ecosystem scene. Note how the effect draws the viewer's eye to the in-focus tree in the center of the image.

The size of the circle of confusion is affected by the radius of the aperture and the distance between the object and the lens. The *focal distance* is the distance from the lens to the plane of objects that project to a zero-radius circle of confusion. These points appear to be perfectly in focus. In practice, objects do not have to be exactly on the focal plane to appear in sharp focus; as long as the circle of confusion is roughly smaller than a pixel,

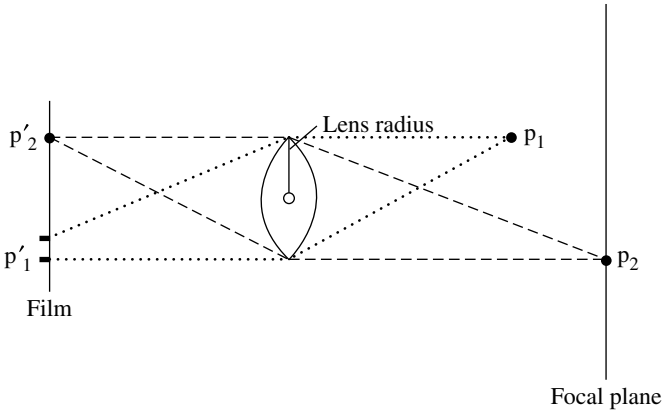
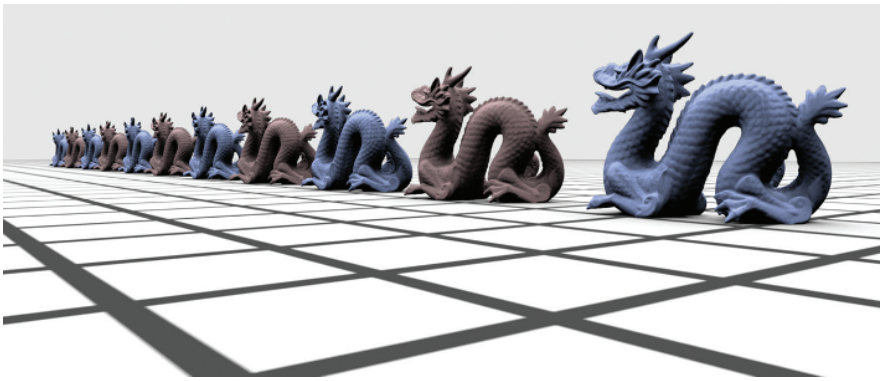
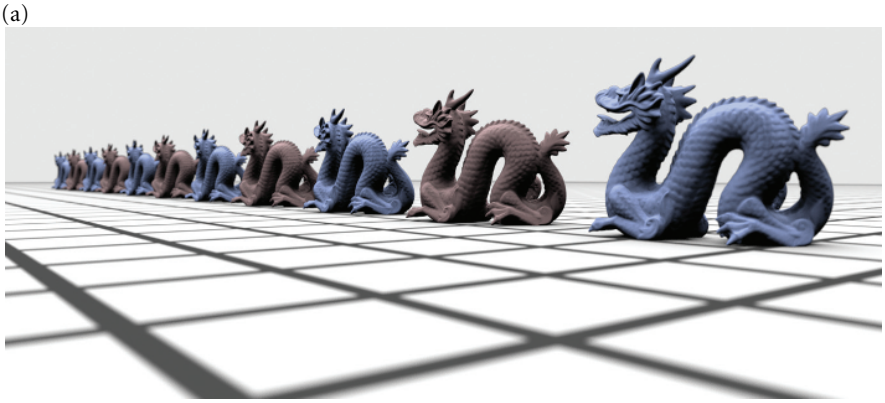


Figure 6.6: Real-world cameras have a lens with finite aperture and lens controls that adjust the lens position with respect to the film plane. Because the aperture is finite, objects in the scene aren't all imaged onto the film in perfect focus. Here, the point p_1 doesn't lie on the plane of points in perfect focus, so it images to an area p'_1 on the film and is blurred. The point p_2 does lie on the focal plane, so it images to a point p'_2 and is in focus. Either increasing the aperture size or increasing an object's distance from the focal plane will increase an out-of-focus object's blurriness.

PerspectiveCamera 310
PerspectiveCamera::
dxCamera 312
PerspectiveCamera::
dyCamera 312
RayDifferential::
rxDirection 69
RayDifferential::rxOrigin 69
RayDifferential::
ryDirection 69
RayDifferential::ryOrigin 69
Vector 57



(a)



(b)

Figure 6.7: (a) Scene rendered with no depth of field and (b) depth of field due to a relatively small lens aperture, which gives only a small amount of blurriness in the out-of-focus regions.

objects appear to be in focus. The range of distances from the lens at which objects appear in focus is called the lens's *depth of field*.

Projective cameras take two extra parameters for depth of field: one sets the size of the lens aperture, and the other sets the focal distance.

⟨*ProjectiveCamera Protected Data*⟩ +≡ 305
```
float lensRadius, focalDistance;
```

⟨*Initialize depth of field parameters*⟩ ≡ 305
```
lensRadius = lensr;
focalDistance = focald;
```

The math behind computing circles of confusion for simple lenses is not difficult; it mostly involves repeated application of similar triangles and some reasonable approximations about the shape of the lens profile. This process can be easily modeled in a ray tracer with just a few lines of code. All that is necessary is to choose a point on the lens and find the appropriate ray that starts on the lens at that point such that objects in the plane

ProjectiveCamera::
focalDistance 314

ProjectiveCamera::
lensRadius 314

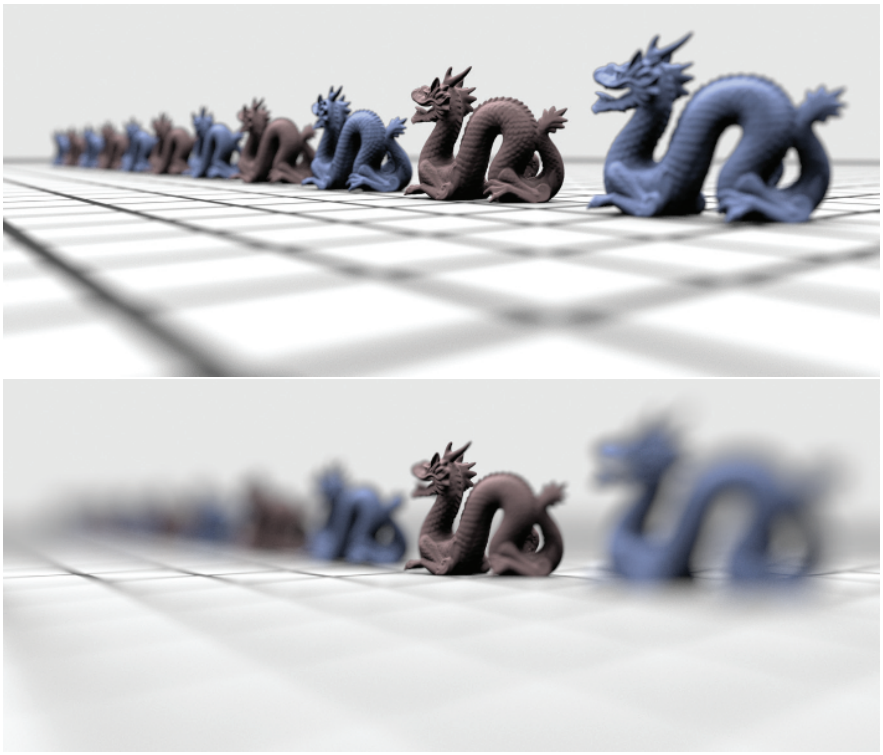


Figure 6.8: As the size of the lens aperture increases, the size of the circle of confusion in the out-of-focus areas increases, giving a greater amount of blur on the image plane.

of focus are in focus (Figure 6.10). It is generally necessary to trace many rays for each image pixel in order to adequately sample the lens for smooth depth of field. Figure 6.11 shows the ecosystem scene from Figure 6.9 with only four samples per pixel (Figure 6.9 had 128 samples per pixel).

⟨*Modify ray for depth of field*⟩ ≡ 309, 312

```
if (lensRadius > 0.) {
    ⟨Sample point on lens 317⟩
    ⟨Compute point on plane of focus 318⟩
    ⟨Update ray for effect of lens 318⟩
}
```

CameraSample 342
CameraSample::lensU 342
CameraSample::lensV 342
ConcentricSampleDisk() 667
ProjectiveCamera::lensRadius 314

The `ConcentricSampleDisk()` function, defined in Chapter 13, takes a (u, v) sample position in $[0, 1]^2$ and maps it to a 2D unit disk centered at the origin $(0, 0)$. To turn this into a point on the lens, these coordinates are scaled by the lens radius. The `CameraSample` class provides the (u, v) lens-sampling parameters in the `CameraSample::lensU` and `CameraSample::lensV` fields.

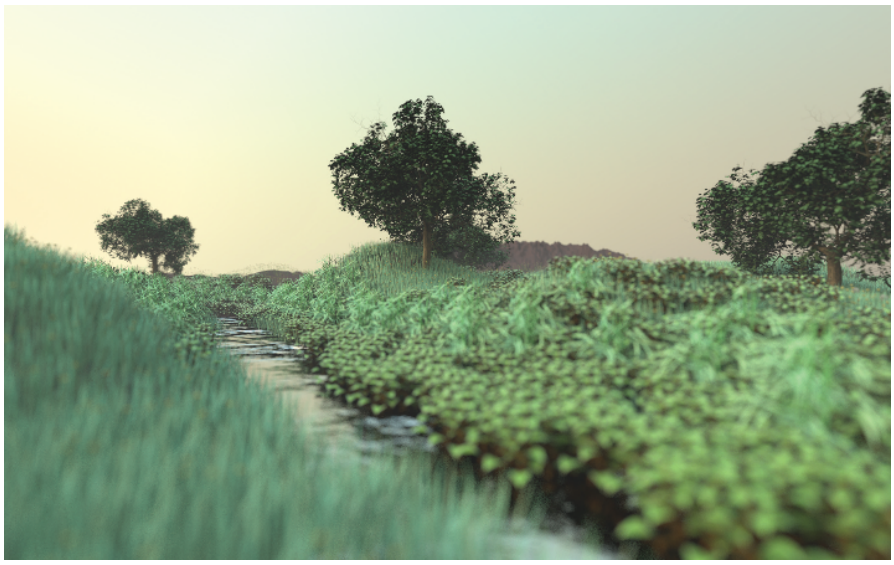


Figure 6.9: Depth of field gives a greater sense of depth and scale to the ecosystem scene.

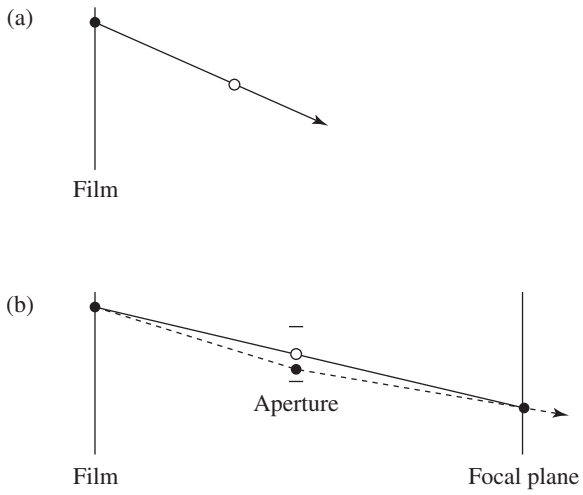


Figure 6.10: (a) For a pinhole camera model, a single camera ray is associated with each point on the film plane (filled circle), given by the ray that passes through the single point of the pinhole lens (empty circle). (b) For a camera model with a finite aperture, we sample a point (filled circle) on the disk-shaped lens for each ray. We can also compute the ray that passes through the center of the lens (corresponding to the pinhole model) and the point where it intersects the focal plane (solid line). We know that all objects in the focal plane must be in focus, regardless of the lens sample position. Therefore, the ray corresponding to the lens position sample (dashed line) is given by the ray starting on the lens sample point and passing through the computed intersection point on the focal plane.

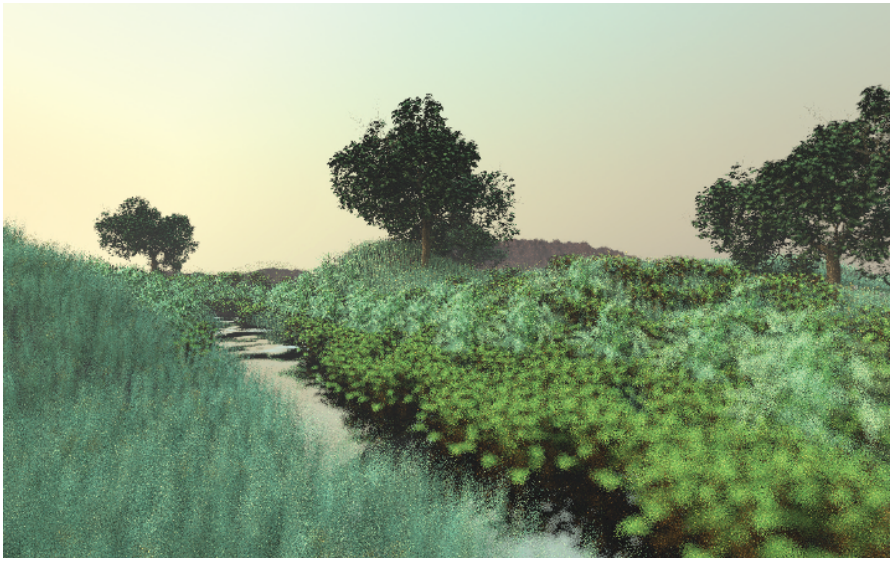


Figure 6.11: Ecosytem scene with depth of field and only four samples per pixel: the depth of field is undersampled and the image is grainy.

⟨*Sample point on lens*⟩ ≡ 315

```
float lensU, lensV;
ConcentricSampleDisk(sample.lensU, sample.lensV, &lensU, &lensV);
lensU *= lensRadius;
lensV *= lensRadius;
```

The ray's origin is this point on the lens. Now it is necessary to determine the proper direction for the new ray. This direction could be computed using Snell's law, which describes how light refracts when passing from one medium (e.g., air) to another (e.g., glass), but the specifics of this particular problem make this simpler.[1] We know that *all* rays from the given image sample through the lens must converge at the same point on the focal plane. Furthermore, we know that rays through the center of the lens are not refracted, so finding the appropriate point of convergence is a matter of intersecting the unperturbed ray from the pinhole model with the focal plane and then setting the ray's direction to be the vector from the point on the lens to the intersection point.

For this simple model, the focal plane is perpendicular to the z axis and the ray starts at the origin, so intersecting the ray through the lens center with the focal plane is straightforward. The t value of the intersection is given by

$$t = \frac{\text{focalDistance}}{\mathbf{d}_z}.$$

CameraSample::lensU 342
CameraSample::lensV 342
ConcentricSampleDisk() 667
ProjectiveCamera::
lensRadius 314

1 We are also assuming a single thin spherical lens element. Simulating complex lens systems with multiple glass elements is much more involved; see the exercises and the "Further Reading" section.

⟨*Compute point on plane of focus*⟩ ≡ 315
```
float ft = focalDistance / ray->d.z;
Point Pfocus = (*ray)(ft);
```

Now the ray can be initialized. The origin is set to the sampled point on the lens and the direction is set so that the ray passes through the point on the plane of focus, `Pfocus`.

⟨*Update ray for effect of lens*⟩ ≡ 315
```
ray->o = Point(lensU, lensV, 0.f);
ray->d = Normalize(Pfocus - ray->o);
```

6.3 ENVIRONMENT CAMERA

One advantage of ray tracing compared to scan line or rasterization-based rendering methods is that it's easy to employ unusual image projections. We have great freedom in how the image sample positions are mapped into ray directions, since the rendering algorithm doesn't depend on properties such as straight lines in the scene always projecting to straight lines in the image.

In this section, we will describe a camera model that traces rays in all directions around a point in the scene, giving a two-dimensional view of everything that is visible from that point. Consider a sphere around the camera position in the scene; choosing points on that sphere gives directions to trace rays in. If we parameterize the sphere with spherical coordinates, each point on the sphere is associated with a (θ, ϕ) pair, where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. (See Section 5.5.2 for more details on spherical coordinates.) This type of image is particularly useful because it represents all of the incident light at a point on the scene. It will be useful later when we discuss environment lighting—a rendering technique that uses image-based representations of light in a scene. Figure 6.12 shows this camera in action with the San Miguel model. θ values range from 0 at the top of the image to π at the bottom of the image, and ϕ values range from 0 to 2π, moving from left to right across the image.[2]

⟨*EnvironmentCamera Declarations*⟩ ≡
```
class EnvironmentCamera : public Camera {
public:
    ⟨EnvironmentCamera Public Methods 319⟩
};
```

The `EnvironmentCamera` derives directly from the `Camera` class, not the `ProjectiveCamera` class. This is because the environmental projection is nonlinear and cannot be captured by a single 4×4 matrix. It is defined in the files `cameras/environment.h` and `cameras/environment.cpp`.

Camera 302
EnvironmentCamera 318
Point 63
ProjectiveCamera 305
ProjectiveCamera::
focalDistance 314
Ray::d 67
Ray::o 67
Vector::Normalize() 63

2 Readers familiar with cartography will recognize this as an equirectangular projection.



Figure 6.12: The San Miguel model rendered with the `EnvironmentCamera`, which traces rays in all directions from the camera position. The resulting image gives a representation of all light arriving at that point in the scene and can be used for the image-based lighting techniques described in Chapters 12 and 15.

⟨*EnvironmentCamera Public Methods*⟩ ≡ 318
```
EnvironmentCamera(const AnimatedTransform &cam2world, float sopen,
                  float sclose, Film *film)
    : Camera(cam2world, sopen, sclose, film) {
}
```

⟨*EnvironmentCamera Method Definitions*⟩ ≡
```
float EnvironmentCamera::GenerateRay(const CameraSample &sample,
                                     Ray *ray) const {
    float time = Lerp(sample.time, shutterOpen, shutterClose);
    ⟨Compute environment camera ray direction 319⟩
    *ray = Ray(Point(0,0,0), dir, 0.f, INFINITY, time);
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

AnimatedTransform 96
Camera 302
Camera::CameraToWorld 302
Camera::shutterClose 302
Camera::shutterOpen 302
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::time 342
EnvironmentCamera 318
Film 403
Film::xResolution 403
Film::yResolution 403
INFINITY 1002
Lerp() 1000
M_PI 1002
Point 63
Ray 66
Vector 57

To compute the (θ, ϕ) coordinates for this ray, NDC coordinates are computed from the raster image sample position and then scaled to cover the (θ, ϕ) range. Next, the spherical coordinate formula is used to compute the ray direction, and finally the direction is converted to world space. (Note that because the y direction is "up" in camera space, here the x and y coordinates in the spherical coordinate formula are exchanged in comparison to usage elsewhere in the system.)

⟨*Compute environment camera ray direction*⟩ ≡ 319
```
float theta = M_PI * sample.imageY / film->yResolution;
float phi = 2 * M_PI * sample.imageX / film->xResolution;
Vector dir(sinf(theta) * cosf(phi), cosf(theta),
           sinf(theta) * sinf(phi));
```

FURTHER READING

In his seminal Sketchpad system, Sutherland (1963) was the first to use projection matrices for computer graphics. Akenine-Möller, Haines, and Hoffman (2008) have provided a particularly well-written derivation of the orthographic and perspective projection matrices in *Real Time Rendering*. Other good references for projections are Rogers and Adams's *Mathematical Elements for Computer Graphics* (1990), Watt and Watt (1992), Foley et al. (1990), and Eberly's book on game engine design (Eberly 2001).

Potmesil and Chakravarty (1981, 1982, 1983) did early work on depth of field and motion blur in computer graphics. Cook and collaborators developed a more accurate model for these effects based on *distribution ray tracing*; this is the approach used for the depth of field calculations in this chapter (Cook, Porter, and Carpenter 1984; Cook 1986).

Kolb, Mitchell, and Hanrahan (1995) investigated simulating complex camera lens systems with ray tracing in order to model the imaging effects of real cameras. Another unusual projection method was used by Greene and Heckbert (1986) for generating images for Omnimax® theaters. The `EnvironmentCamera` in this chapter is similar to the camera model described by Musgrave (1992).

EXERCISES

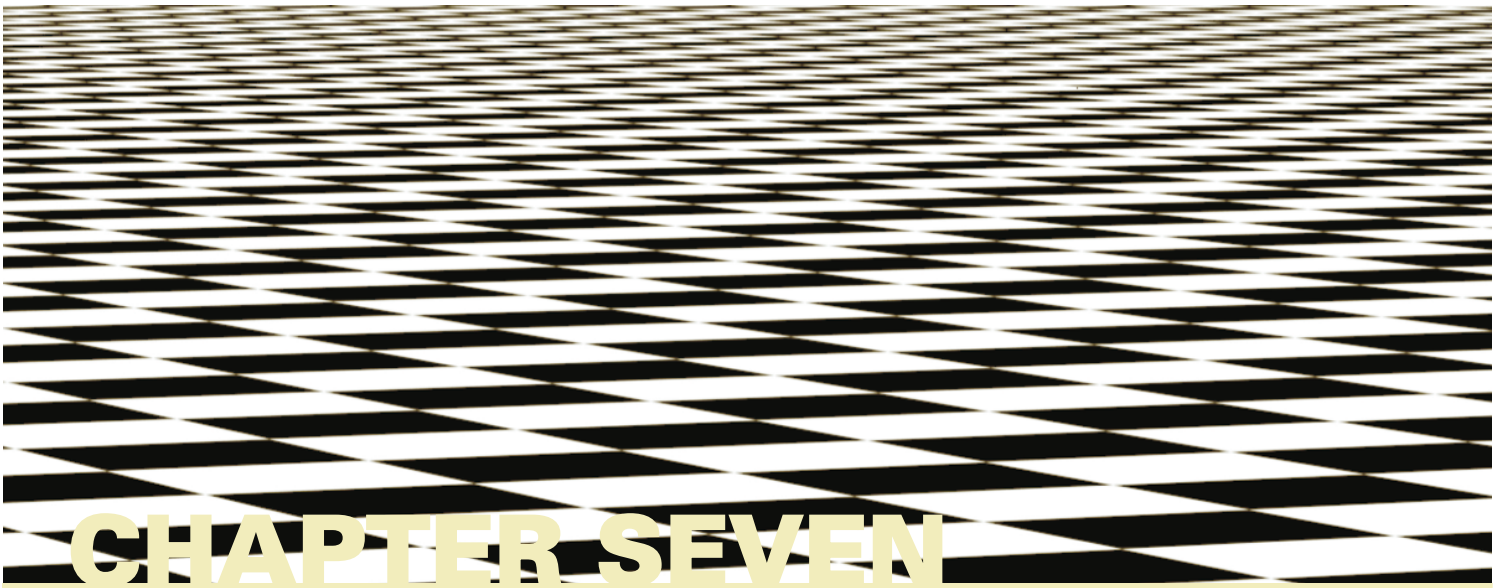
6.1 Modify the way that time samples are generated in the camera implementations in this chapter to support different shutter models. For example, many cameras expose the film by sliding a rectangular slit across the film. This leads to interesting effects when objects are moving in a different direction than the exposure slit (Glassner 1999; Stephenson 2006). Implement this slit shutter model for both horizontal and vertical slits, and also a leaf shutter. Create scenes that clearly show the effects of each shutter type.

6.2 The standard model for depth of field in computer graphics models the circle of confusion as imaging a point in the scene to a circle with uniform intensity, although many real lenses produce circles of confusion with nonlinear variation such as a Gaussian distribution. This effect is known as "Bokeh" (Buhler and Wexler 2002). For example, catadioptric (mirror) lenses produce doughnut-shaped highlights when small points of light are viewed out of focus. Modify the implementation of depth of field in `pbrt` to produce images with this effect (for example, by biasing the distribution of lens sample positions). Render images showing the difference between this and the standard model.

6.3 Write an application that loads images rendered by the `EnvironmentCamera` and uses texture mapping to apply them to a sphere centered at the eyepoint such that they can be viewed interactively. The user should be able to freely change the viewing direction. If the appropriate texture-mapping function is used for generating texture coordinates on the sphere, the image generated by the application will appear as if the viewer was at the camera's location in the scene when it was rendered, thus giving the user the ability to interactively look around the scene.

EnvironmentCamera 318

● 6.4 Kolb, Mitchell, and Hanrahan (1995) have described a camera model for ray tracing based on simulating the lens system of a real camera, which is comprised of a set of glass lenses arranged to form an image on the film plane. Read their paper and implement a camera model in `pbrt` that implements their algorithm for following rays through lens systems. Test your implementation with some of the lens description data from their paper.



CHAPTER SEVEN

07 SAMPLING AND RECONSTRUCTION

Although the final output of a renderer like `pbrt` is a two-dimensional grid of colored pixels, incident radiance is actually a continuous function defined over the film plane. The manner in which the discrete pixel values are computed from this continuous function can noticeably affect the quality of the final image generated by the renderer; if this process is not performed carefully, artifacts will be present. Fortunately, a relatively small amount of additional computation to this end can substantially improve the quality of the rendered images.

This chapter introduces *sampling theory*—the theory of taking discrete sample values from functions defined over continuous domains and then using those samples to reconstruct new functions that are similar to the original. Building on principles of sampling theory, the `Sampler`s in this chapter select sample points on the image plane at which incident radiance will be computed (recall that in the previous chapter `Camera`s used `Sample`s generated by a `Sampler` to construct their camera rays). Five `Sampler` implementations are described in this chapter, spanning a variety of approaches to the sampling problem. This chapter concludes with the `Filter` class and the `Film` class. The `Filter` is used to determine how multiple samples near each pixel are blended together to compute the final pixel value, and `Film` implementations filter image samples and store them in images.

7.1 SAMPLING THEORY

A digital image is represented as a set of pixel values, typically aligned on a rectangular grid. When a digital image is displayed on a physical device, these values are used to set the intensities and colors of pixels on the display. When thinking about digital images, it is important to differentiate between image pixels, which represent the value of a function at a particular sample location, and display pixels, which are physical objects

that emit light with some distribution. For example, in a CRT, each phosphor glows with a distribution that falls off from its center. Pixel intensity has angular distribution as well; it is mostly uniform for CRTs, although it can be quite directional on LCD displays. Displays use the image pixel values to construct a new image function over the display surface. This function is defined at all points on the display, not just the infinitesimal points of the digital image's pixels. This process of taking a collection of sample values and converting them back to a continuous function is called *reconstruction*.

In order to compute the discrete pixel values in the digital image, it is necessary to sample the original continuously defined image function. In `pbrt`, like most other ray-tracing renderers, the only way to get information about the image function is to sample it by tracing rays. For example, there is no general method that can compute bounds on the variation of the image function between two points on the film plane. While an image could be generated by just sampling the function precisely at the pixel positions, a better result can be obtained by taking more samples at different positions and incorporating this additional information about the image function into the final pixel values. Indeed, for the best-quality result, the pixel values should be computed such that the reconstructed image on the display device is as close as possible to the original image of the scene on the virtual camera's film plane. Note that this is a subtly different goal than expecting the display's pixels to take on the image function's actual value at their positions. Handling this difference is the main goal of the algorithms implemented in this chapter.[1]

Because the sampling and reconstruction process involves approximation, it introduces error known as *aliasing,* which can manifest itself in many ways, including jagged edges or flickering in animations. These errors occur because the sampling process is not able to capture all of the information from the continuously defined image function.

As an example of these ideas, consider a one-dimensional function (which we will interchangeably refer to as a signal), given by $f(x)$, where we can evaluate $f(x')$ at any desired location x' in the function's domain. Each such x' is called a *sample position*, and the value of $f(x')$ is the *sample value*. Figure 7.1 shows a set of samples of a smooth 1D function, along with a reconstructed signal $\tilde{f}$ that approximates the original function f. In this example, $\tilde{f}$ is a piecewise linear function that approximates f by linearly interpolating neighboring sample values (readers already familiar with sampling theory will recognize this as reconstruction with a hat function). Because the only information available about f comes from the sample values at the positions x', $\tilde{f}$ is unlikely to match f perfectly since there is no information about f's behavior between the samples.

Fourier analysis can be used to evaluate the quality of the match between the reconstructed function and the original. This section will introduce the main ideas of Fourier analysis with enough detail to work through some parts of the sampling and reconstruction processes, but will omit proofs of many properties and skip details that aren't directly

1 In this book, we will ignore the subtle issues related to the characteristics of the physical display pixels and will work under the assumption that the display performs the ideal reconstruction process described later in this section. This assumption is patently at odds with how actual displays work, but it avoids unnecessary complicating of the analysis here. Chapter 3 of Glassner (1995) has a good treatment of nonidealized display devices and their impact on the image sampling and reconstruction process.

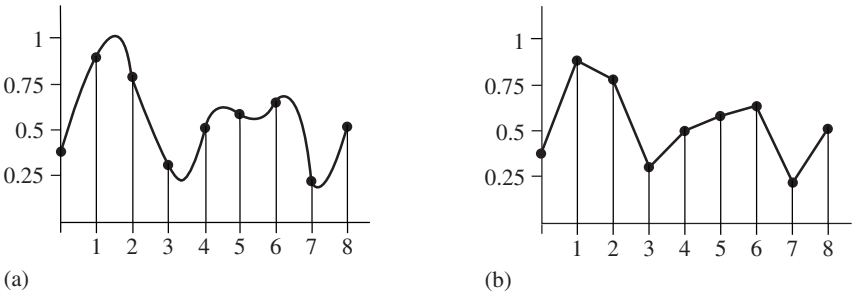


Figure 7.1: (a) By taking a set of *point samples* of $f(x)$ (indicated by black dots), we determine the value of the function at those positions. (b) The sample values can be used to *reconstruct* a function $\tilde{f}(x)$ that is an approximation to $f(x)$. The sampling theorem, introduced in Section 7.1.3, makes a precise statement about the conditions on $f(x)$, the number of samples taken, and the reconstruction technique used under which $\tilde{f}(x)$ is exactly the same as $f(x)$. The fact that the original function can sometimes be reconstructed exactly from point samples alone is remarkable.

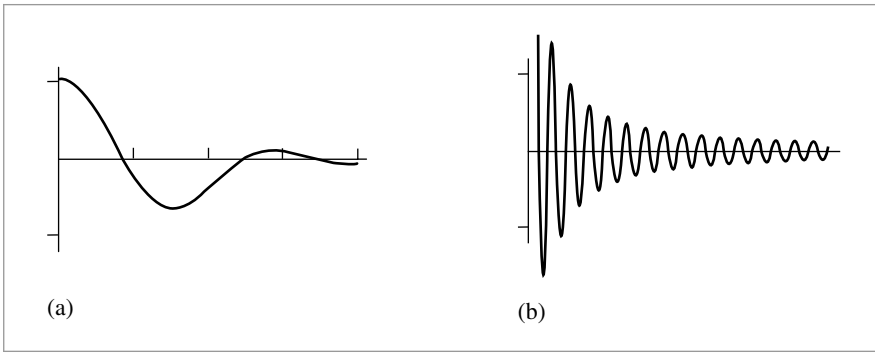


Figure 7.2: (a) Low-frequency function, and (b) high-frequency function. Roughly speaking, the higher frequency a function is, the more quickly it varies over a given region.

relevant to the sampling algorithms used in `pbrt`. The "Further Reading" section of this chapter has pointers to more detailed information about these topics.

7.1.1 THE FREQUENCY DOMAIN AND THE FOURIER TRANSFORM

One of the foundations of Fourier analysis is the Fourier transform, which represents a function in the *frequency domain*. (We will say that functions are normally expressed in the *spatial domain*.) Consider the two functions graphed in Figure 7.2. The function in Figure 7.2(a) varies relatively slowly as a function of x, while the function in Figure 7.2(b) varies much more rapidly. The slower-varying function is said to have lower frequency content. Figure 7.3 shows the frequency space representations of these two functions; the lower-frequency function's representation goes to zero more quickly than the higher-frequency function.

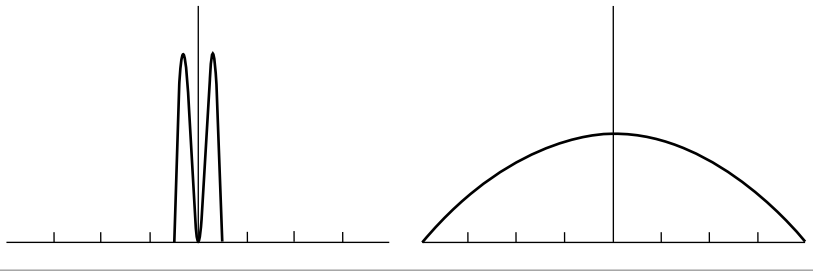


Figure 7.3: Frequency Space Representations of the Functions in Figure 7.2. The graphs show the contribution of each frequency ω to each of the functions in the spatial domain.

Most functions can be decomposed into a weighted sum of shifted sinusoids. This remarkable fact was first described by Joseph Fourier, and the Fourier transform converts a function into this representation. This frequency space representation of a function gives insight into some of its characteristics—the distribution of frequencies in the sine functions corresponds to the distribution of frequencies in the original function. Using this form, it is possible to use Fourier analysis to gain insight into the error that is introduced by the sampling and reconstruction process, and how to reduce the perceptual impact of this error.

The Fourier transform of a one-dimensional function $f(x)$ is[2]

$$F(\omega) = \int_{-\infty}^{\infty} f(x) \mathrm{e}^{-i2\pi\omega x} \mathrm{d}x. \tag{7.1}$$

(Recall that $\mathrm{e}^{ix} = \cos x + i \sin x$, where $i = \sqrt{-1}$.) For simplicity, here we will consider only *even* functions where $f(-x) = f(x)$, in which case the Fourier transform of f has no imaginary terms. The new function F is a function of *frequency*, ω.[3] We will denote the Fourier transform operator by $\mathcal{F}$, such that $\mathcal{F}\{f(x)\} = F(\omega)$. $\mathcal{F}$ is clearly a linear operator—that is, $\mathcal{F}\{af(x)\} = a\mathcal{F}\{f(x)\}$ for any scalar a, and $\mathcal{F}\{f(x) + g(x)\} = \mathcal{F}\{f(x)\} + \mathcal{F}\{g(x)\}$.

Equation (7.1) is called the *Fourier analysis* equation, or sometimes just the *Fourier transform*. We can also transform from the frequency domain back to the spatial domain using the *Fourier synthesis* equation, or the *inverse Fourier transform*:

$$f(x) = \int_{-\infty}^{\infty} F(\omega) \mathrm{e}^{i2\pi\omega x} \, \mathrm{d}\omega. \tag{7.2}$$

2 The reader should be warned that the constants in front of these integrals are not always the same in different fields. For example, some authors (including many in the physics community) prefer to multiply both integrals by $1/\sqrt{2\pi}$.

3 In this chapter, we will use the ω symbol to denote frequency. Throughout the rest of the book, ω denotes normalized direction vectors. This overloading of notation should never be confusing, given the contexts where these symbols are used. Similarly, when we refer to a function's "spectrum," we are referring to its distribution of frequencies in its frequency space representation, rather than anything related to color.

Table 7.1: Fourier Pairs. Functions in the spatial domain and their frequency space representations. Because of the symmetry properties of the Fourier transform, if the left column is instead considered to be frequency space, then the right column is the spatial equivalent of those functions as well.

Spatial Domain	Frequency Space Representation
Box: $f(x) = 1$ if $\|x\| < 1/2$, 0 otherwise	Sinc: $f(\omega) = \text{sinc}(\omega) = \sin(\pi\omega)/(\pi\omega)$
Gaussian: $f(x) = e^{-\pi x^2}$	Gaussian: $f(\omega) = e^{-\pi\omega^2}$
Constant: $f(x) = 1$	Delta: $f(\omega) = \delta(\omega)$
Sinusoid: $f(x) = \cos x$	Translated delta: $f(\omega) = \pi(\delta(1/2 - \omega) + \delta(1/2 + \omega))$
Shah: $f(x) = \text{III}_T(x) = T \sum_i \delta(x - Ti)$	Shah: $f(\omega) = \text{III}_{1/T}(\omega) = (1/T) \sum_i \delta(\omega - i/T)$

Table 7.1 shows a number of important functions and their frequency space representations. A number of these functions are based on the Dirac delta distribution, a special function that is defined such that $\int \delta(x)\mathrm{d}x = 1$, and for all $x \neq 0$, $\delta(x) = 0$. An important consequence of these properties is that

$$\int f(x)\delta(x)\,\mathrm{d}x = f(0).$$

The delta distribution cannot be expressed as a standard mathematical function, but instead is generally thought of as the limit of a unit area box function centered at the origin with width approaching zero.

7.1.2 IDEAL SAMPLING AND RECONSTRUCTION

Using frequency space analysis, we can now formally investigate the properties of sampling. Recall that the sampling process requires us to choose a set of equally spaced sample positions and compute the function's value at those positions. Formally, this corresponds to multiplying the function by a "shah," or "impulse train," function, an infinite sum of equally spaced delta functions. The shah $\text{III}_T(x)$ is defined as

$$\text{III}_T(x) = T \sum_{i=-\infty}^{\infty} \delta(x - iT),$$

where T defines the period, or *sampling rate*. This formal definition of sampling is illustrated in Figure 7.4. The multiplication yields an infinite sequence of values of the function at equally spaced points:

$$\text{III}_T(x) f(x) = T \sum_i \delta(x - iT) f(iT).$$

These sample values can be used to define a reconstructed function $\tilde{f}$ by choosing a reconstruction filter function $r(x)$ and computing the *convolution*

$$\big(\text{III}_T(x) f(x)\big) \otimes r(x),$$

where the convolution operation $\otimes$ is defined as

$$f(x) \otimes g(x) = \int_{-\infty}^{\infty} f(x')g(x - x')\,\mathrm{d}x'.$$

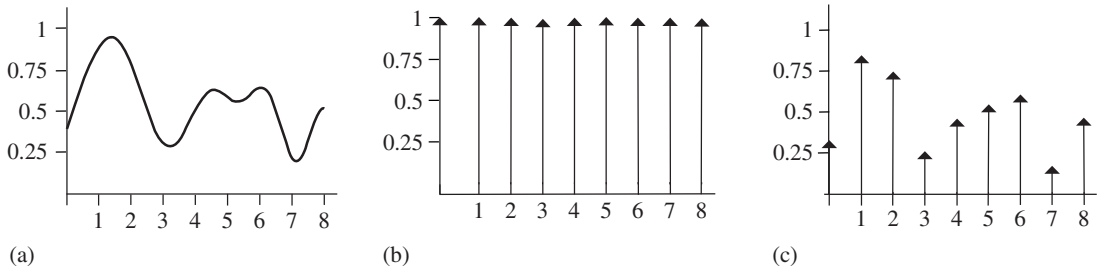


Figure 7.4: Formalizing the Sampling Process. (a) The function $f(x)$ is multiplied by (b) the shah function $\text{III}_T(x)$, giving (c) an infinite sequence of scaled delta functions that represent its value at each sample point.

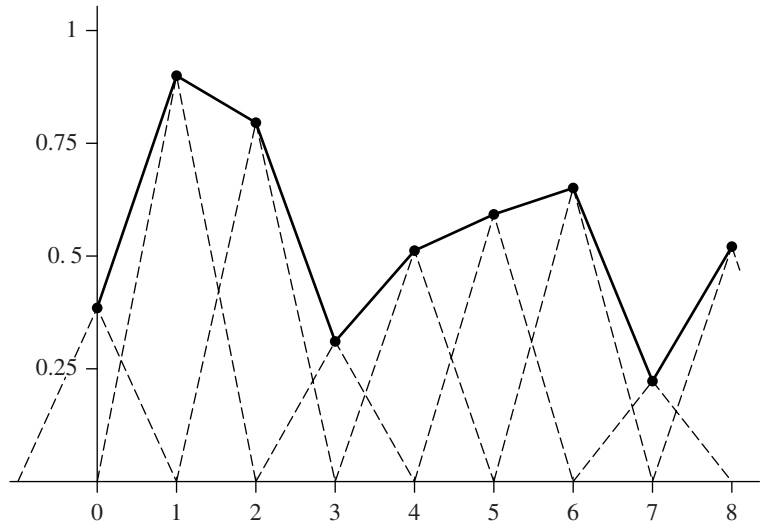


Figure 7.5: The sum of instances of the triangle reconstruction filter, shown with dashed lines, gives the reconstructed approximation to the original function, shown with a solid line.

For reconstruction, convolution gives a weighted sum of scaled instances of the reconstruction filter centered at the sample points:

$$\tilde{f}(x) = T \sum_{i=-\infty}^{\infty} f(iT)r(x - iT).$$

For example, in Figure 7.1, the triangle reconstruction filter, $f(x) = \max(0, 1 - |x|)$, was used. Figure 7.5 shows the scaled triangle functions used for that example.

We have gone through a process that may seem gratuitiously complex in order to end up at an intuitive result: the reconstructed function $\tilde{f}(x)$ can be obtained by interpolating among the samples in some manner. By setting up this background carefully, however, Fourier analysis can now be applied to the process more easily.

We can gain a deeper understanding of the sampling process by analyzing the sampled function in the frequency domain. In particular, we will be able to determine the conditions under which the original function can be exactly recovered from its values at the sample locations—a very powerful result. For the discussion here, we will assume for now that the function $f(x)$ is *band limited*—there exists some frequency ω_0 such that $f(x)$ contains no frequencies greater than ω_0. By definition, band-limited functions have frequency space representations with compact support, such that $F(\omega) = 0$ for all $|\omega| > \omega_0$. Both of the spectra in Figure 7.3 are band limited.

An important idea used in Fourier analysis is the fact that the Fourier transform of the product of two functions $\mathcal{F}\{f(x)g(x)\}$ can be shown to be the convolution of their individual Fourier transforms $F(\omega)$ and $G(\omega)$:

$$\mathcal{F}\{f(x)g(x)\} = F(\omega) \otimes G(\omega).$$

It is similarly the case that convolution in the spatial domain is equivalent to multiplication in the frequency domain:

$$\mathcal{F}\{f(x) \otimes g(x)\} = F(\omega)G(\omega). \tag{7.3}$$

These properties are derived in the standard references on Fourier analysis. Using these ideas, the original sampling step in the spatial domain, where the product of the shah function and the original function $f(x)$ is found, can be equivalently described by the convolution of $F(\omega)$ with another shah function in frequency space.

We also know the spectrum of the shah function $\text{III}_T(x)$ from Table 7.1; the Fourier transform of a shah function with period T is another shah function with period $1/T$. This reciprocal relationship between periods is important to keep in mind: it means that if the samples are farther apart in the spatial domain, they are closer together in the frequency domain.

Thus, the frequency domain representation of the sampled signal is given by the convolution of $F(\omega)$ and this new shah function. Convolving a function with a delta function just yields a copy of the function, so convolving with a shah function yields an infinite sequence of copies of the original function, with spacing equal to the period of the shah (Figure 7.6). This is the frequency space representation of the series of samples.

Now that we have this infinite set of copies of the function's spectrum, how do we reconstruct the original function? Looking at Figure 7.6, the answer is obvious: just discard all of the spectrum copies except the one centered at the origin, giving the original $F(\omega)$. In order to throw away all but the center copy of the spectrum, we multiply by a box function of the appropriate width (Figure 7.7). The box function $\Pi_T(x)$ of width T is defined as

$$\Pi_T(x) = \begin{cases} 1/(2T) & |x| < T \\ 0 & \text{otherwise.} \end{cases}$$

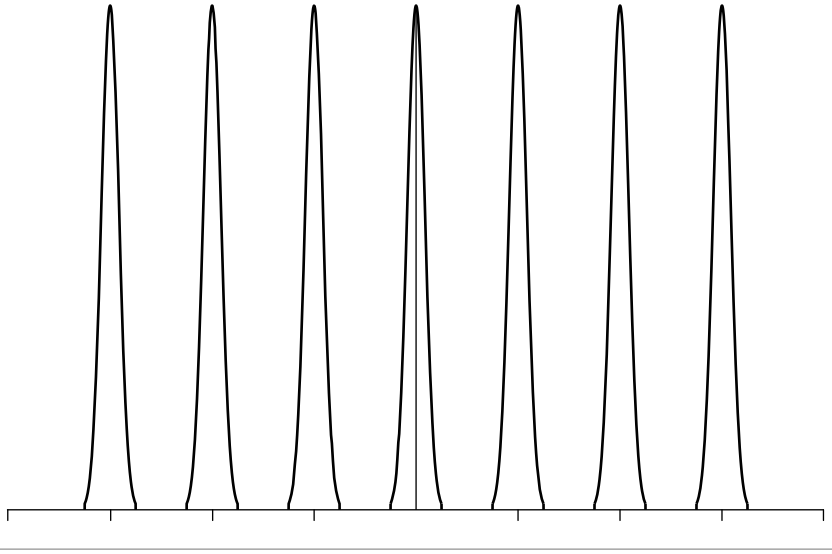


Figure 7.6: The Convolution of $F(\omega)$ and the Shah Function. The result is infinitely many copies of F.

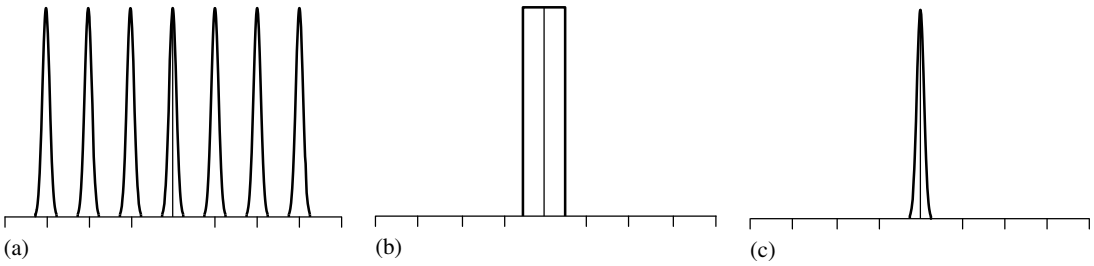


Figure 7.7: Multiplying (a) a series of copies of $F(\omega)$ by (b) the appropriate box function yields (c) the original spectrum.

This multiplication step corresponds to convolution with the reconstruction filter in the spatial domain. This is the ideal sampling and reconstruction process. To summarize:

$$\tilde{F} = \left(F(\omega) \otimes \text{III}_{1/T}(\omega)\right) \Pi_T(x).$$

This is a remarkable result: we have been able to determine the exact frequency space representation of $f(x)$, purely by sampling it at a set of regularly spaced points. Other than knowing that the function was band limited, no additional information about the composition of the function was used.

Applying the equivalent process in the spatial domain will likewise recover $f(x)$ exactly. Because the inverse Fourier transform of the box function is the sinc function, ideal reconstruction in the spatial domain is found by

$$\tilde{f} = \big(f(x)\mathrm{III}_T(x)\big) \otimes \mathrm{sinc}(x),$$

or

$$\tilde{f}(x) = \sum_{i=-\infty}^{\infty} \mathrm{sinc}(x-i)\, f(i).$$

Unfortunately, because the sinc function has infinite extent, it is necessary to use all of the sample values $f(i)$ to compute any particular value of $\tilde{f}(x)$ in the spatial domain. Filters with finite spatial extent are preferable for practical implementations even though they don't reconstruct the original function perfectly.

A commonly used alternative in graphics is to use the box function for reconstruction, effectively averaging all of the sample values within some region around x. This is a very poor choice, as can be seen by considering the box filter's behavior in the frequency domain: This technique attempts to isolate the central copy of the function's spectrum by *multiplying by a sinc*, which not only does a bad job of selecting the central copy of the function's spectrum but includes high-frequency contributions from the infinite series of other copies of it as well.

7.1.3 ALIASING

In addition to the sinc function's infinite extent, one of the most serious practical problems with the ideal sampling and reconstruction approach is the assumption that the signal is band limited. For signals that are not band limited, or signals that aren't sampled at a sufficiently high sampling rate for their frequency content, the process described earlier will reconstruct a function that is different than the original signal.

The key to successful reconstruction is the ability to exactly recover the original spectrum $F(\omega)$ by multiplying the sampled spectrum with a box function of the appropriate width. Notice that in Figure 7.6, the copies of the signal's spectrum are separated by empty space, so perfect reconstruction is possible. Consider what happens, however, if the original function was sampled with a lower sampling rate. Recall that the Fourier transform of a shah function III_T with period T is a new shah function with period $1/T$. This means that if the spacing between samples increases in the spatial domain, the sample spacing decreases in the frequency domain, pushing the copies of the spectrum $F(\omega)$ closer together. If the copies get too close together, they start to overlap.

Because the copies are added together, the resulting spectrum no longer looks like many copies of the original (Figure 7.8). When this new spectrum is multiplied by a box function, the result is a spectrum that is similar but not equal to the original $F(\omega)$: High-frequency details in the original signal leak into lower-frequency regions of the spectrum of the reconstructed signal. These new low-frequency artifacts are called *aliases* (because high frequencies are "masquerading" as low frequencies), and the resulting signal is

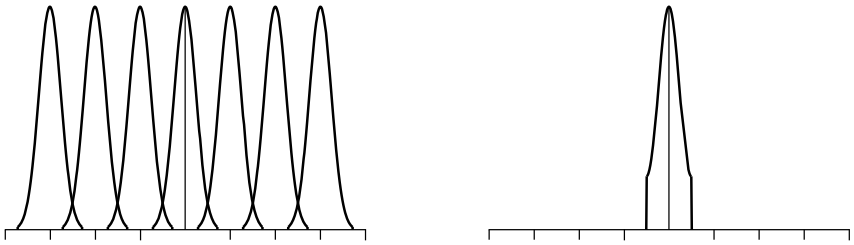


Figure 7.8: (a) When the sampling rate is too low, the copies of the function's spectrum overlap, resulting in (b) aliasing when reconstruction is performed.

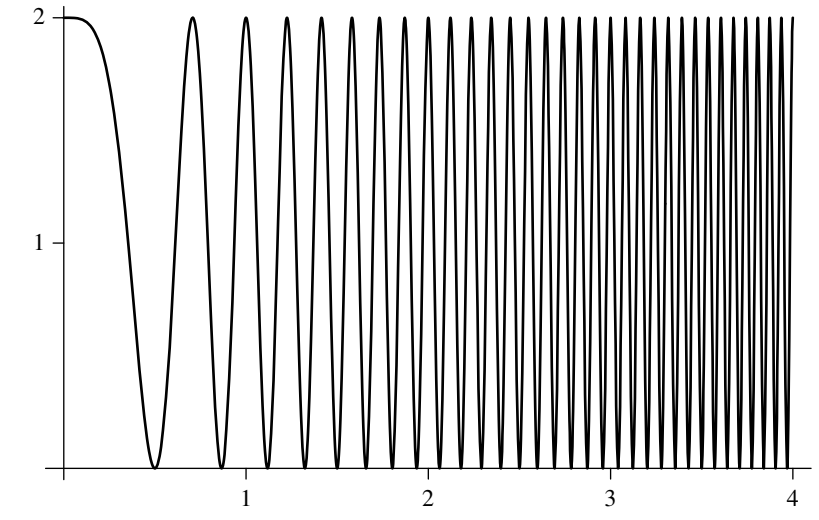
said to be *aliased*. Figure 7.9 shows the effects of aliasing from undersampling and then reconstructing the one-dimensional function $f(x) = 1 + \cos(4x^2)$.

A possible solution to the problem of overlapping spectra is to simply increase the sampling rate until the copies of the spectrum are sufficiently far apart to not overlap, thereby eliminating aliasing completely. In fact, the *sampling theorem* tells us exactly what rate is required. This theorem says that as long as the frequency of uniform sample points ω_s is greater than twice the maximum frequency present in the signal ω_0, it is possible to reconstruct the original signal perfectly from the samples. This minimum sampling frequency is called the *Nyquist frequency*.

For signals that are not band limited ($\omega_0 = \infty$), it is impossible to sample at a high enough rate to perform perfect reconstruction. Non-band-limited signals have spectra with infinite support, so no matter how far apart the copies of their spectra are (i.e., how high a sampling rate we use), there will always be overlap. Unfortunately, few of the interesting functions in computer graphics are band limited. In particular, any function containing a discontinuity cannot be band limited, and therefore we cannot perfectly sample and reconstruct it. This makes sense because the function's discontinuity will always fall between two samples and the samples provide no information about the location of the discontinuity. Thus, it is necessary to apply different methods besides just increasing the sampling rate in order to counteract the error that aliasing can introduce to the renderer's results.

7.1.4 ANTIALIASING TECHNIQUES

If one is not careful about sampling and reconstruction, myriad artifacts can appear in the final image. It is sometimes useful to distinguish between artifacts due to sampling and those due to reconstruction; when we wish to be precise we will call sampling artifacts *prealiasing* and reconstruction artifacts *postaliasing*. Any attempt to fix these errors is broadly classified as *antialiasing*. This section reviews a number of antialiasing techniques beyond just increasing the sampling rate everywhere.



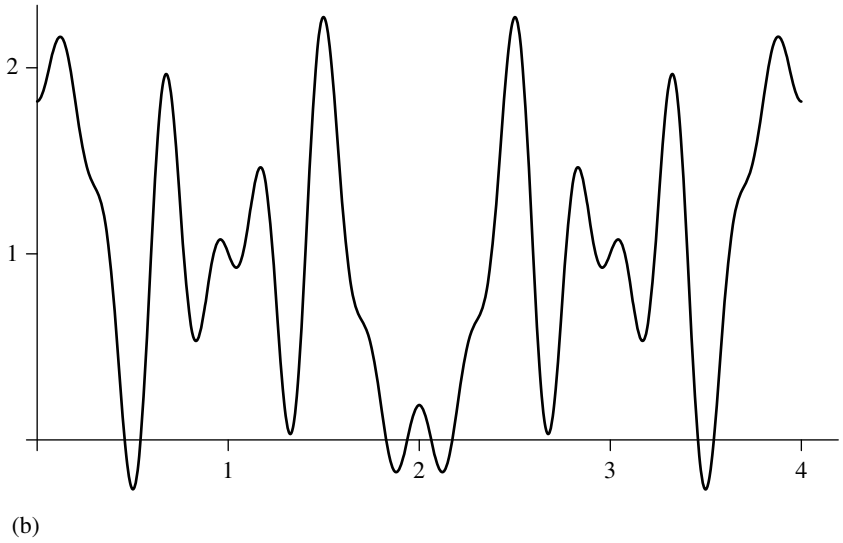


Figure 7.9: Aliasing from Point Sampling the Function $1 + \cos(4x^2)$. (a) The function. (b) The reconstructed function from sampling it with samples spaced 0.125 units apart and performing perfect reconstruction with the sinc filter. Aliasing causes the high-frequency information in the original function to be lost and to reappear as lower-frequency error.

Nonuniform Sampling

Although the image functions that we will be sampling are known to have infinite-frequency components and thus can't be perfectly reconstructed from point samples, it is possible to reduce the visual impact of aliasing by varying the spacing between samples in a nonuniform way. If ξ denotes a random number between zero and one, a nonuniform set of samples based on the impulse train is

$$\sum_{i=-\infty}^{\infty} \delta\left(x - \left(iT + \frac{1}{2} - \xi\right)\right).$$

For a fixed sampling rate that isn't sufficient to capture the function, both uniform and nonuniform sampling produce incorrect reconstructed signals. However, nonuniform sampling tends to turn the regular aliasing artifacts into noise, which is less distracting to the human visual system. In frequency space, the copies of the sampled signal end up being randomly shifted as well, so that when reconstruction is performed the result is random error rather than coherent aliasing.

Adaptive Sampling

Another approach that has been suggested to combat aliasing is *adaptive supersampling:* if we can identify the regions of the signal with frequencies higher than the Nyquist limit, we can take additional samples in those regions without needing to incur the computational expense of increasing the sampling frequency everywhere. It can be difficult to get this approach to work well in practice, because finding all of the places where supersampling is needed is difficult. Most techniques for doing so are based on examining adjacent sample values and finding places where there is a significant change in value between the two; the assumption is that the signal has high frequencies in that region.

In general, adjacent sample values cannot tell us with certainty what is really happening between them: Even if the values are the same, the functions may have huge variation between them. Alternatively, adjacent samples may have substantially different values without any aliasing actually being present. For example, the texture-filtering algorithms in Chapter 10 work hard to eliminate aliasing due to image maps and procedural textures on surfaces in the scene; we would not want an adaptive sampling routine to needlessly take extra samples in an area where texture values are changing quickly but no excessively high frequencies are actually present.

Prefiltering

Another approach to eliminating aliasing that sampling theory offers is to filter (i.e., blur) the original function so that no high frequencies remain that can't be captured accurately at the sampling rate being used. This approach is applied in the texture functions of Chapter 10. While this technique changes the character of the function being sampled by removing information from it, it is generally less objectionable than aliasing.

Recall that we would like to multiply the original function's spectrum with a box filter with width chosen so that frequencies above the Nyquist limit are removed. In the spatial domain, this corresponds to convolving the original function with a sinc filter,

$$f(x) \otimes \operatorname{sinc}(2\omega_s x).$$

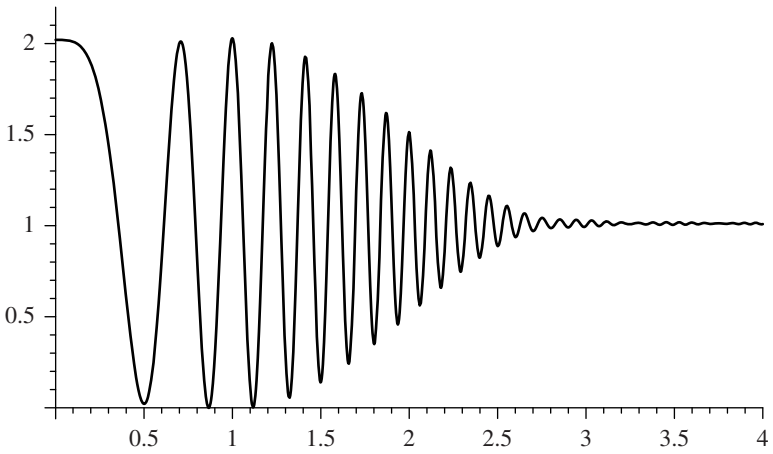


Figure 7.10: Graph of the function $1 + \cos(4x^2)$ convolved with a filter that removes frequencies beyond the Nyquist limit for a sampling rate of $T = .125$. High-frequency detail has been removed from the function, so that the new function can at least be sampled and reconstructed without aliasing.

In practice, we can use a filter with finite extent that works well. The frequency space representation of this filter can help clarify how well it approximates the behavior of the ideal sinc filter.

Figure 7.10 shows the function $1 + \cos(4x^2)$ convolved with a variant of the sinc with finite extent that will be introduced in Section 7.7. Note that the high-frequency details have been eliminated; this function can be sampled and reconstructed at the sampling rate used in Figure 7.9 without aliasing.

7.1.5 APPLICATION TO IMAGE SYNTHESIS

The application of these ideas to the two-dimensional case of sampling and reconstructing images of rendered scenes is straightforward: we have an image, which we can think of as a function of two-dimensional (x, y) image locations to radiance values L:

$$f(x, y) \rightarrow L.$$

The good news is that, with our ray tracer, we can evaluate this function at any (x, y) point that we choose. The bad news is that it's not generally possible to prefilter f to remove the high frequencies from it before sampling. Therefore, the samplers in this chapter will use both strategies of increasing the sampling rate beyond the basic pixel spacing in the final image as well as nonuniformly distributing the samples to turn aliasing into noise.

It is useful to generalize the definition of the scene function to a higher-dimensional function that also depends on the time t and (u, v) lens position at which it is sampled. Because the rays from the camera are based on these five quantities, varying any of them

gives a different ray and thus a potentially different value of f. For a particular image position, the radiance at that point will generally vary across both time (if there are moving objects in the scene) and position on the lens (if the camera has a finite-aperture lens).

Even more generally, because many of the integrators defined in Chapters 15 and 16 use statistical techniques to estimate the radiance along a given ray, they may return a different radiance value when repeatedly given the same ray. If we further extend the scene radiance function to include sample values used by the integrator (e.g., values used to choose points on area light sources for illumination computations), we have an even higher-dimensional image function

$$f(x, y, t, u, v, i_1, i_2, \ldots) \rightarrow L.$$

Sampling all of these dimensions well is an important part of generating high-quality imagery efficiently. For example, if we ensure that nearby (x, y) positions on the image tend to have dissimilar (u, v) positions on the lens, the resulting rendered images will have less error because each sample is more likely to account for information about the scene that its neighboring samples do not. The `Sampler` classes in the next few sections will address the issue of sampling all of these dimensions as well as possible.

7.1.6 SOURCES OF ALIASING IN RENDERING

Geometry is one of the most common causes of aliasing in rendered images. When projected onto the image plane, an object's boundary introduces a step function—the image function's value instantaneously jumps from one value to another. Not only do step functions have infinite frequency content as mentioned earlier, but, even worse, the perfect reconstruction filter causes artifacts when applied to aliased samples: ringing artifacts appear in the reconstructed function, an effect known as the *Gibbs phenomenon*. Figure 7.11 shows an example of this effect for a 1D function. Choosing an effective reconstruction filter in the face of aliasing requires a mix of science, artistry, and personal taste, as we will see later in this chapter.

Very small objects in the scene can also cause geometric aliasing. If the geometry is small enough that it falls between samples on the image plane, it can unpredictably disappear and reappear over multiple frames of an animation.

Another source of aliasing can come from the texture and materials on an object. *Shading aliasing* can be caused by texture maps that haven't been filtered correctly (addressing this problem is the topic of much of Chapter 10), or from small highlights on shiny surfaces. If the sampling rate is not high enough to sample these features adequately, aliasing will result. Furthermore, a sharp shadow cast by an object introduces another step function in the final image. While it is possible to identify the position of step functions from geometric edges on the image plane, detecting step functions from shadow boundaries is more difficult.

The key insight about aliasing in rendered images is that we can never remove all of its sources, so we must develop techniques to mitigate its impact on the quality of the final image.

Sampler 340

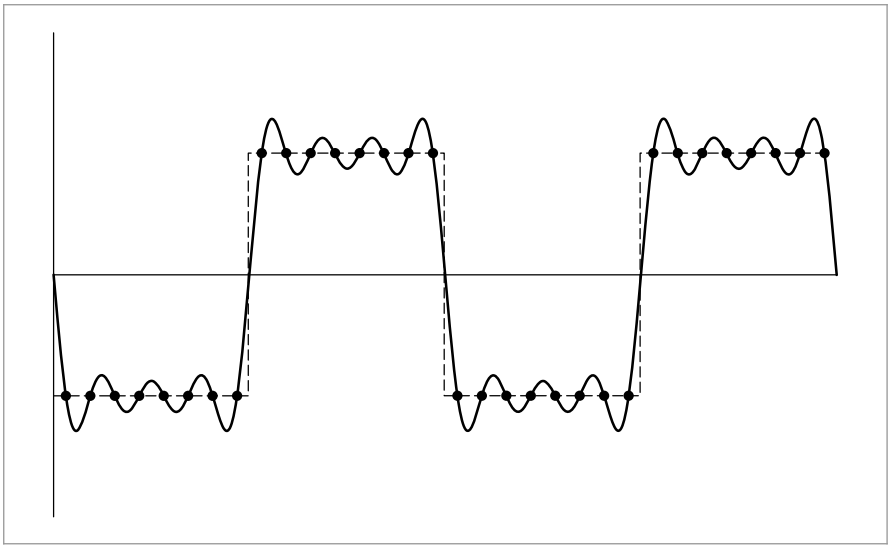


Figure 7.11: Illustration of the Gibbs Phenomenon. When a function hasn't been sampled at the Nyquist rate and the set of aliased samples is reconstructed with the sinc filter, the reconstructed function will have "ringing" artifacts, where it oscillates around the true function. Here a 1D step function (dashed line) has been sampled with a sample spacing of 0.125. When reconstructed with the sinc, the ringing appears (solid line).

7.1.7 UNDERSTANDING PIXELS

There are two ideas about pixels that are important to keep in mind throughout the remainder of this chapter. First, it is crucial to remember that the pixels that constitute an image are point samples of the image function at discrete points on the image plane; there is no "area" associated with a pixel. As Alvy Ray Smith (1995) has emphatically pointed out, thinking of pixels as small squares with finite area is an incorrect mental model that leads to a series of errors. By introducing the topics of this chapter with a signal processing approach, we have tried to lay the groundwork for a more accurate mental model.

The second issue is that the pixels in the final image are naturally defined at discrete integer (x, y) coordinates on a pixel grid, but the `Samplers` in this chapter generate image samples at continuous floating-point (x, y) positions. The natural way to map between these two domains is to round continuous coordinates to the nearest discrete coordinate; this is appealing since it maps continuous coordinates that happen to have the same value as discrete coordinates to that discrete coordinate. However, the result is that given a set of discrete coordinates spanning a range $[x_0, x_1]$, the set of continuous coordinates that covers that range is $[x_0 - 1/2, x_1 + 1/2)$. Thus, any code that generates continuous sample positions for a given discrete pixel range is littered with 1/2 offsets. It is easy to forget some of these, leading to subtle errors.

Sampler 340

If we instead truncate continuous coordinates c to discrete coordinates d by

$$d = \lfloor c \rfloor,$$

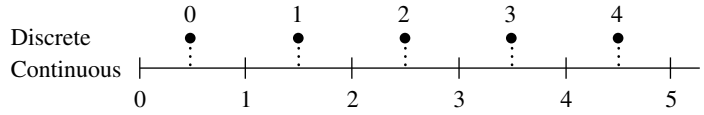


Figure 7.12: Pixels in an image can be addressed with either *discrete* or *continuous* coordinates. A discrete image four pixels wide covers the continuous pixel range $[0, 5)$. A particular discrete pixel d's coordinate in the continuous representation is $d + 1/2$.

and convert from discrete to continuous by

$$c = d + 1/2,$$

then the range of continuous coordinates for the discrete range $[x_0, x_1]$ is naturally $[x_0, x_1 + 1)$ and the resulting code is much simpler (Heckbert 1990a). This convention, which we will adopt in `pbrt`, is shown graphically in Figure 7.12.

7.2 IMAGE SAMPLING INTERFACE

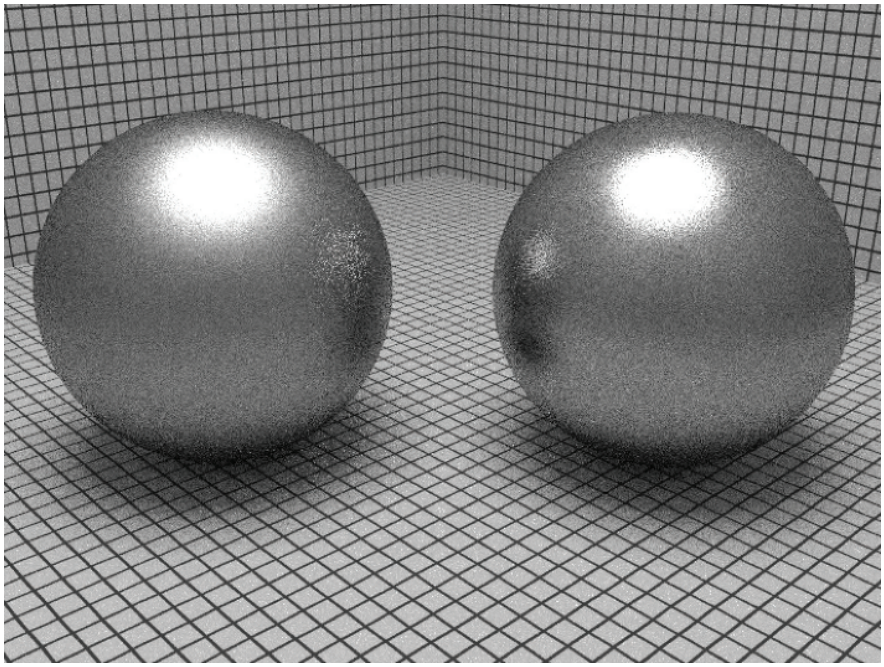
We can now describe the operation of a few classes that generate good image sampling patterns. It may be surprising to see that some of them have a significant amount of complexity behind them. In practice, creating good sample patterns can substantially improve a ray tracer's efficiency, allowing it to create a high-quality image with fewer rays than if a lower-quality pattern was used. Because the run time expense for using the best sampling patterns is approximately the same as for lower-quality patterns, and because evaluating the radiance for each image sample is expensive, doing this work pays dividends (Figure 7.13).

The core sampling declarations and functions are in the files `core/sampler.h` and `core/sampler.cpp`. Each of the sample generation implementations is in its own source file in the `samplers/` directory.

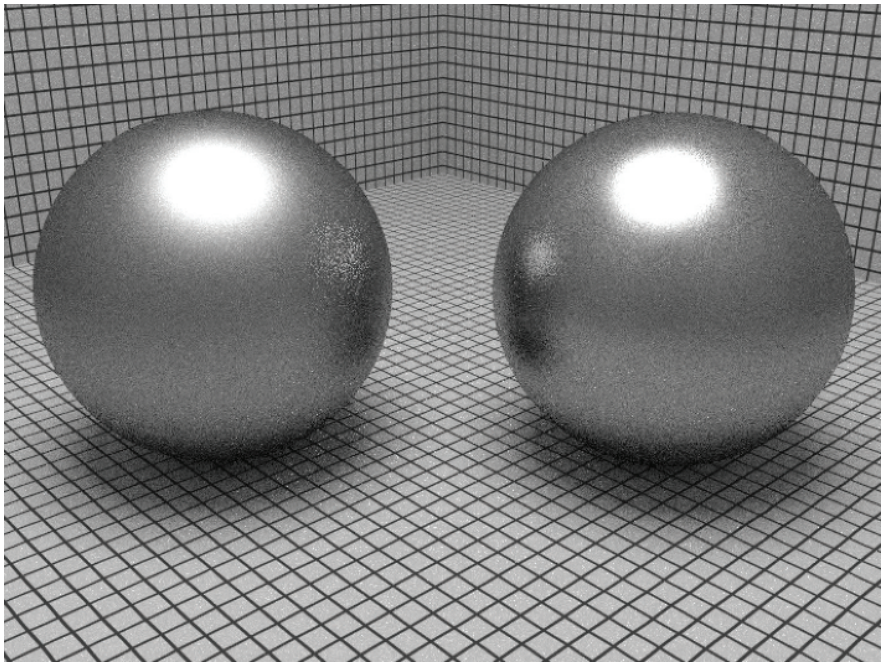
All of the sampler implementations inherit from an abstract `Sampler` class that defines their interface. The task of `Sampler`s is to generate a sequence of multidimensional sample positions. Two dimensions give the raster space image sample position, and another gives the time at which the sample should be taken; this ranges from zero to one, and is scaled by the camera to cover the time period that the shutter is open. Two more sample values give a (u, v) lens position for depth of field; these also vary from zero to one.

Just as well-placed sample points can help conquer the complexity of the 2D image function, most of the light transport algorithms in Chapters 15 and 16 use sample points for tasks like choosing positions on area light sources when estimating illumination. Choosing these points is also the job of the `Sampler`, since it is able to take the sample points chosen for adjacent image samples into account when selecting samples at new points. Doing so can improve the quality of the results of the light transport algorithms.

`Sampler` 340



(a)



(b)

Figure 7.13: Scene rendered with (a) a relatively ineffective sampler and (b) a carefully designed sampler, using the same number of samples for each. The improvement in image quality, ranging from the edges of the highlights to the quality of the glossy reflections, is noticeable.

⟨*Sampling Declarations*⟩ ≡
```
class Sampler {
public:
    ⟨Sampler Interface 340⟩
    ⟨Sampler Public Data 340⟩
protected:
    ⟨Sampler Protected Methods⟩
};
```

All of the Sampler implementations take a few common parameters that must be passed to the base class's constructor. These include the overall image resolution in the x and y dimensions, the number of samples the implementation expects to generate for each pixel in the final image, and the range of time over which the camera's shutter is open. These values are stored in member variables for later use.

⟨*Sampler Method Definitions*⟩ ≡
```
Sampler::Sampler(int xstart, int xend, int ystart, int yend, int spp,
                 float sopen, float sclose)
    : xPixelStart(xstart), xPixelEnd(xend), yPixelStart(ystart),
      yPixelEnd(yend), samplesPerPixel(spp), shutterOpen(sopen),
      shutterClose(sclose) { }
```

The Sampler implementation should generate samples for pixels with x coordinates ranging from xPixelStart to xPixelEnd-1, inclusive, and analogously for y coordinates. The time values for samples should similarly be in the range shutterOpen to shutterClose.

⟨*Sampler Public Data*⟩ ≡ 340
```
const int xPixelStart, xPixelEnd, yPixelStart, yPixelEnd;
const int samplesPerPixel;
const float shutterOpen, shutterClose;
```

Samplers must implement the Sampler::GetMoreSamples() method, which is a pure virtual function. The SamplerRendererTask::Run() method calls this function until it returns zero, which signifies that all samples have been generated. Otherwise, it generates one or more samples, returning the number of samples generated and filling in sample values in the array pointed to by the sample parameter. All of the dimensions of the sample values it generates should have values in the range [0, 1], except for CameraSample::imageX and CameraSample::imageY, which are specified with respect to the image size in raster coordinates.

The caller also passes a RNG for the sampler to use for any random number generation it needs to do; Samplers could equivalently store their own RNGs, though code that calls this method generally already has one available.

⟨*Sampler Interface*⟩ ≡ 340
```
virtual int GetMoreSamples(Sample *sample, RNG &rng) = 0;
```

The MaximumSampleCount() method returns the maximum number of sample values that the Sampler will ever return from its implementation of the GetMoreSamples() method. It allows the renderer to preallocate memory for the array of Samples passed into GetMoreSamples().

CameraSample::imageX 342
RNG 1003
Sample 343
Sampler 340
Sampler::
GetMoreSamples() 340
Sampler::samplesPerPixel 340
Sampler::shutterClose 340
Sampler::shutterOpen 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340
SamplerRendererTask::Run() 30

⟨*Sampler Interface*⟩ +≡ 340
```
virtual int MaximumSampleCount() = 0;
```

Samplers may implement the ReportResults() method; it allows the renderer to report back to the sampler which rays were generated, what radiance values were computed, and the intersection points found for a collection of samples originally from GetMoreSamples(). The sampler may use this information for adaptive sampling algorithms, deciding to take more samples close to the ones that were returned here.

The return value indicates whether or not the sample values should be added to the image being generated. For some adaptive sampling algorithms, the sampler may want to cause an initial collection of samples (and their results) to be discarded, generating a new set to replace them completely. Because most of the Samplers in this chapter do not implement adaptive sampling, a default implementation of this method just returns true.

⟨*Sampler Method Definitions*⟩ +≡
```
bool Sampler::ReportResults(Sample *samples, const RayDifferential *rays,
        const Spectrum *Ls, const Intersection *isects, int count) {
    return true;
}
```

Recall from Section 1.3.4 that the SamplerRenderer refines the task of rendering the complete image into a number of tasks that can execute in parallel. The Sampler::GetSubSampler() method is a key part of this process. As the scene description file is parsed, a single Sampler is created; conceptually, its responsibility is to generate samples for the entire image. The GetSubSampler() method returns a new Sampler that is responsible for generating samples for a subset of the image; its num parameter ranges from 0 to count-1, where count is the total number of subsamplers being used. Implementations of this method must ensure that the aggregate set of generated samples by all of the subsamplers covers the image; how they decompose the image is not defined.

⟨*Sampler Interface*⟩ +≡ 340
```
virtual Sampler *GetSubSampler(int num, int count) = 0;
```

Most implementations of the GetSubSampler() method decompose the image into rectangular tiles and have each subsampler generate samples for a single tile. ComputeSubWindow() is a utility function that computes a pixel sampling range given a tile number num and a total number of tiles count.

⟨*Sampler Method Definitions*⟩ +≡
```
void Sampler::ComputeSubWindow(int num, int count, int *newXStart,
        int *newXEnd, int *newYStart, int *newYEnd) const {
    ⟨Determine how many tiles to use in each dimension, nx and ny 342⟩
    ⟨Compute x and y pixel sample range for sub-window 342⟩
}
```

Intersection 186
RayDifferential 69
Sample 343
Sampler 340
SamplerRenderer 25
Spectrum 263

It's desirable that the shapes of image tiles be roughly square (as opposed to, for example, being one pixel tall and many pixels wide). This leads to more coherent access to the scene data in memory and potentially to better performance. Starting with the total number of tiles in count, the number of tiles in the x dimension is successively halved while the number in y is doubled until nx is an odd number or the number of x tiles times

the y image resolution is less than or equal to the number of y tiles times the x image resolution. (When nx is odd, we can no longer halve it and have nx * ny == count.) This is a reasonable effort to make square tiles; a more complex implementation probably isn't worthwhile.

⟨*Determine how many tiles to use in each dimension,* nx *and* ny⟩ ≡ 341
```
int dx = xPixelEnd - xPixelStart, dy = yPixelEnd - yPixelStart;
int nx = count, ny = 1;
while ((nx & 0x1) == 0 && 2 * dx * ny < dy * nx) {
    nx >>= 1;
    ny <<= 1;
}
```

Given the number of tiles in each direction, the integer tile number (xo, yo) is easily computed and hence the pixel range.

⟨*Compute x and y pixel sample range for sub-window*⟩ ≡ 341
```
int xo = num % nx, yo = num / nx;
float tx0 = float(xo) / float(nx), tx1 = float(xo+1) / float(nx);
float ty0 = float(yo) / float(ny), ty1 = float(yo+1) / float(ny);
*newXStart = Floor2Int(Lerp(tx0, xPixelStart, xPixelEnd));
*newXEnd   = Floor2Int(Lerp(tx1, xPixelStart, xPixelEnd));
*newYStart = Floor2Int(Lerp(ty0, yPixelStart, yPixelEnd));
*newYEnd   = Floor2Int(Lerp(ty1, yPixelStart, yPixelEnd));
```

7.2.1 SAMPLE REPRESENTATION AND ALLOCATION

The Sample structure is used by Samplers to store a single sample. After one or more Samples is initialized by a call to the Sampler's GetMoreSamples() method, the Sampler RendererTask::Run() method passes each of the Samples to the camera and integrators, which read values from it to construct the camera ray and perform lighting calculations.

Sample inherits from the CameraSample structure; the CameraSample represents just the sample values that are needed for generating camera rays. This separation allows us to just pass a CameraSample to the Camera::GenerateRay() method, not exposing all of the details of the rest of the Sample structure to Cameras.

⟨*Sampling Declarations*⟩ +≡
```
struct CameraSample {
    float imageX, imageY;
    float lensU, lensV;
    float time;
};
```

Depending on the details of the light transport algorithm being used, different integrators may have different sampling needs beyond the camera sample values. For example, the WhittedIntegrator doesn't do any random sampling, so it doesn't need any additional sample values beyond the ones to generate the camera ray. The DirectLighting

Camera::GenerateRay() 303
Floor2Int() 1002
Lerp() 1000
Sample 343
Sampler 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340
SamplerRendererTask::Run() 30
WhittedIntegrator 42

`Integrator` uses values from the `Sampler` to randomly choose a light source to sample illumination from, as well as to randomly choose positions on area light sources. Therefore, the integrators are given an opportunity to request additional sample values in various quantities. Information about these requirements is stored in the `Sample` object. When it is later passed to the particular `Sampler` implementation, it is the `Sampler`'s responsibility to generate all of the requested types of samples.

⟨*Sampling Declarations*⟩ +≡
```
struct Sample : public CameraSample {
    ⟨Sample Public Methods 344⟩
    ⟨Sample Public Data 344⟩
private:
    ⟨Sample Private Methods⟩
};
```

The `Sample` constructor calls the `Integrator::RequestSamples()` methods of the surface and volume integrators to find out what samples they will need. The integrators can ask for multiple one-dimensional and/or two-dimensional sampling patterns, each with an arbitrary number of entries. For example, in a scene with two area light sources, where the integrator traces four shadow rays to the first source and eight to the second, the integrator would ask for two 2D sample patterns for each image sample, with four and eight samples each. A 2D pattern is required because two dimensions are needed to parameterize the surface of a light. Similarly, if the integrator wanted to randomly select a single light source out of many, it could request a 1D sample with a single value for this purpose and use its `float` value to randomly choose a light.

By informing the `Sampler` of as much of its random sampling needs as possible, the `Integrator` allows the `Sampler` to carefully construct sample points that cover the entire high-dimensional sample space well. For example, the final image is generally better when neighboring image samples tend to sample different positions on the area lights for their illumination computations, allowing more information to be discovered.

In `pbrt`, we don't allow integrators to request 3D or higher-dimensional sample patterns because these are generally not needed for the types of rendering algorithms implemented here. If necessary, an integrator can combine points from lower-dimensional patterns to get higher-dimensional sample points (e.g., a 1D and a 2D sample pattern of the same size can form a 3D pattern). Although this may not be as good as generating a 3D sample pattern directly, it is usually works well in practice. If absolutely necessary, the integrator can always generate a 3D sample pattern itself, though with less ability to easily ensure that nearby image samples sample different regions of the 3D space.

The integrators' implementations of the `Integrator::RequestSamples()` method in turn call the `Sample::Add1D()` and `Sample::Add2D()` methods, which request another sample sequence with a given number of sample values. After they are done calling these methods, the `Sample` constructor can continue, allocating storage for the requested sample values.

CameraSample 342
DirectLightingIntegrator 742
Integrator::
RequestSamples() 740
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sampler 340

⟨*Sample Method Definitions*⟩ ≡
```
Sample::Sample(Sampler *sampler, SurfaceIntegrator *surf,
               VolumeIntegrator *vol, const Scene *scene) {
    if (surf) surf->RequestSamples(sampler, this, scene);
    if (vol)  vol->RequestSamples(sampler, this, scene);
    AllocateSampleMemory();
}
```

The implementations of the `Sample::Add1D()` and `Sample::Add2D()` methods record the number of samples asked for in an array and return an index that the integrator can later use to access the desired sample values in the `Sample`.

⟨*Sample Public Methods*⟩ ≡ 343
```
uint32_t Add1D(uint32_t num) {
    n1D.push_back(num);
    return n1D.size()-1;
}
```

⟨*Sample Public Methods*⟩ +≡ 343
```
uint32_t Add2D(uint32_t num) {
    n2D.push_back(num);
    return n2D.size()-1;
}
```

Most `Samplers` can do a better job of generating particular quantities of these additional samples than others. For example, the `LDSampler` can generate extremely good patterns, although they must have a size that is a power of two. The `Sampler::RoundSize()` method helps communicate this information. Integrators should call this method with the desired number of samples to be taken, giving the `Sampler` an opportunity to adjust the number of samples to a more convenient one. The integrator should then use the returned value as the number of samples to request from the `Sampler`.

⟨*Sampler Interface*⟩ +≡ 340
```
virtual int RoundSize(int size) const = 0;
```

It is the `Sampler`'s responsibility to store the samples it generates for the integrators in the `Sample::oneD` and `Sample::twoD` arrays. For 1D sample patterns, it needs to generate `n1D.size()` independent patterns, where the `i`th pattern has `n1D[i]` sample values. These values are stored in `oneD[i][0]` through `oneD[i][n1D[i]-1]`.

⟨*Sample Public Data*⟩ ≡ 343
```
vector<uint32_t> n1D, n2D;
float **oneD, **twoD;
```

To access the samples, the integrator stores the sample tag returned by `Add1D()` in a member variable (for example, `sampleOffset`) and can then access the sample values in a loop like

Integrator::
RequestSamples() 740
LDSampler 373
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sample::
AllocateSampleMemory() 345
Sample::n1D 344
Sample::n2D 344
Sampler 340
Sampler::RoundSize() 344
Scene 22
SurfaceIntegrator 740
VolumeIntegrator 876

```
for (i = 0; i < sample->n1D[sampleOffset]; ++i) {
    float s = sample->oneD[sampleOffset][i];
    .
    .
    .
}
```

In 2D, the process is equivalent, but the `i`th sample is given by the two values `twoD[offset][2*i]` and `twoD[offset][2*i+1]`.

After all sample requests have been made, the `AllocateSampleMemory()` method allocates space for their storage.

⟨*Sample Method Definitions*⟩ +≡

```
void Sample::AllocateSampleMemory() {
    ⟨Allocate storage for sample pointers 345⟩
    ⟨Compute total number of sample values needed 345⟩
    ⟨Allocate storage for sample values 346⟩
}
```

This method first allocates memory to store the pointers. Rather than performing two allocations, it does a single allocation that gives enough memory for the pointers for both the `oneD` and `twoD` sample arrays. `twoD` is then set to point at an appropriate offset into this memory, after the last pointer for `oneD`. Splitting up a single allocation like this is useful because it ensures that `oneD` and `twoD` point to nearby locations in memory, which is likely to reduce cache misses.

⟨*Allocate storage for sample pointers*⟩ ≡ 345

```
int nPtrs = n1D.size() + n2D.size();
if (!nPtrs) {
    oneD = twoD = NULL;
    return;
}
oneD = AllocAligned<float *>(nPtrs);
twoD = oneD + n1D.size();
```

It then uses the same trick to allocate memory for the actual sample values so that they are contiguous in memory as well. First, it finds the total number of `float` values needed.

⟨*Compute total number of sample values needed*⟩ ≡ 345

```
int totSamples = 0;
for (uint32_t i = 0; i < n1D.size(); ++i)
    totSamples += n1D[i];
for (uint32_t i = 0; i < n2D.size(); ++i)
    totSamples += 2 * n2D[i];
```

AllocAligned() 1013
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344

The method can now allocate a single chunk of memory and hand it out in pieces for the various collections of samples.

⟨*Allocate storage for sample values*⟩ ≡ 345
```
float *mem = AllocAligned<float>(totSamples);
for (uint32_t i = 0; i < n1D.size(); ++i) {
    oneD[i] = mem;
    mem += n1D[i];
}
for (uint32_t i = 0; i < n2D.size(); ++i) {
    twoD[i] = mem;
    mem += 2 * n2D[i];
}
```

The `Sample` destructor, not shown here, just frees the dynamically allocated memory.

A unique instance of the `Sample` structure is created for each rendering task in Section 1.3.4; the housekeeping for doing so is handled by the `Sample::Duplicate()` method. Most of the work involves allocating memory for storing sample values. The implementation is straightforward and not included here.

⟨*Sample Public Methods*⟩ +≡ 343
```
Sample *Duplicate(int count) const;
```

7.3 STRATIFIED SAMPLING

The first sample generator that we will introduce divides the image plane into rectangular regions and generates a single sample inside each region. These regions are commonly called *strata*, and this sampler is called the `StratifiedSampler`. The key idea behind stratification is that by subdividing the sampling domain into nonoverlapping regions and taking a single sample from each one, we are less likely to miss important features of the image entirely, since the samples are guaranteed not to all be close together. Put another way, it does us no good if many samples are taken from nearby points in the sample space, since each new sample doesn't add much new information about the behavior of the image function. From a signal processing viewpoint, we are implicitly defining an overall sampling rate such that the smaller the strata are, the more of them we have, and thus the higher the sampling rate.

The stratified sampler places each sample at a random point inside each stratum by *jittering* the center point of the stratum by a random amount up to half the stratum's width and height. The nonuniformity that results from this jittering helps turn aliasing into noise, as discussed in Section 7.1. The sampler also offers a nonjittered mode, which gives uniform sampling in the strata; this mode is mostly useful for comparisons between different sampling techniques rather than for rendering final images.

Figure 7.14 shows a comparison of a few sampling patterns. The first is a completely random sampling pattern: we generated a number of samples at random without using the strata at all. The result is a terrible sampling pattern; some regions have few samples and other areas have clumps of many samples. The second is a uniform stratified pattern. In the last, the uniform pattern has been jittered, with a random offset added to each sample's location, keeping it inside its cell. This gives a better overall distribution than

AllocAligned() 1013
Sample 343
Sample::Duplicate() 346
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
StratifiedSampler 349

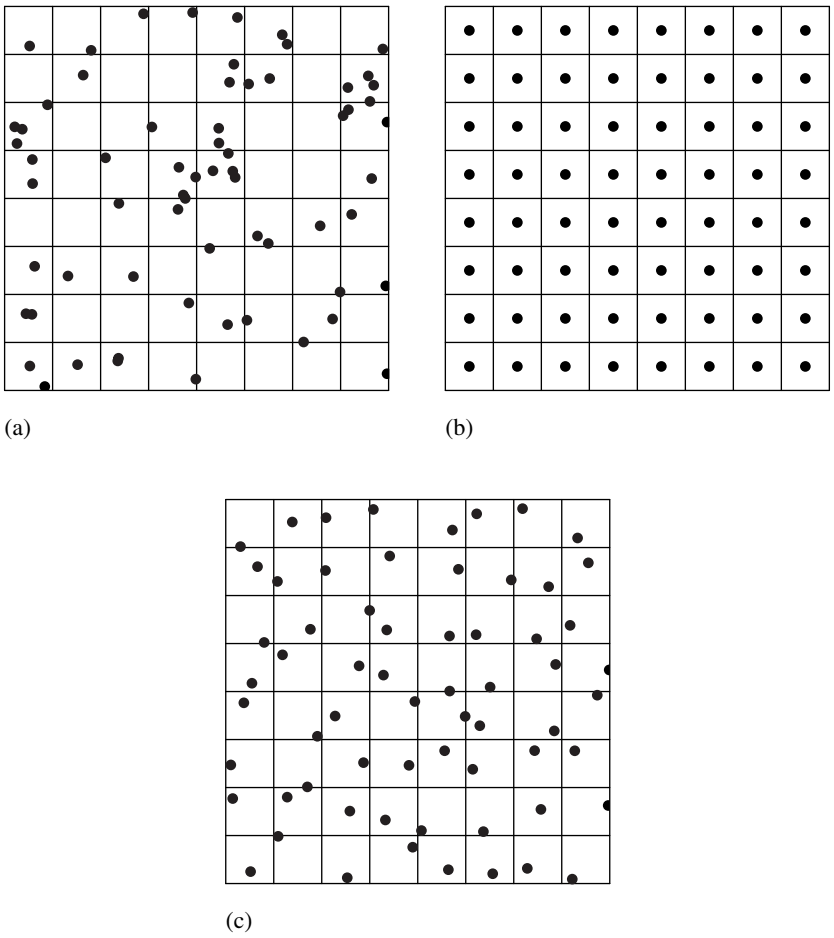


Figure 7.14: Three 2D Sampling Patterns. (a) The random pattern is an ineffective pattern, with many clumps of samples that leave large sections of the image poorly sampled. (b) A uniform stratified pattern is better distributed but can exacerbate aliasing artifacts. (c) A stratified jittered pattern turns aliasing from the uniform pattern into high-frequency noise while still maintaining the benefits of stratification.

the purely random pattern while preserving the benefits of stratification, though there are still some clumps of samples and some regions that are undersampled. We will present more sophisticated image sampling methods in the next two sections that ameliorate some of these remaining shortcomings. Figure 7.15 shows images rendered using the `StratifiedSampler` and shows how jittered sample positions turn aliasing artifacts into less objectionable noise.

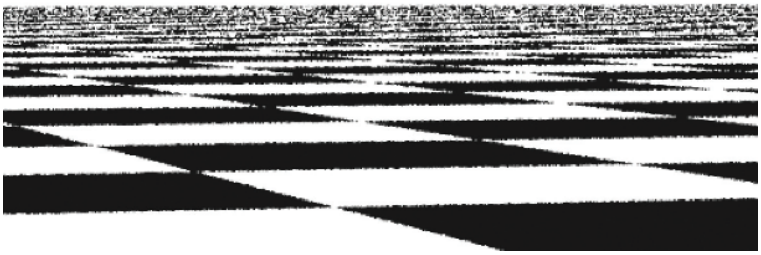
StratifiedSampler 349



(a)



(b)



(c)



(d)

Figure 7.15: Comparison of Image Sampling Methods with a Checkerboard Texture. This is a difficult image to render well, since the checkerboard's frequency with respect to the pixel spacing tends toward infinity as we approach the horizon. (a) A reference image, rendered with 256 samples per pixel, showing something close to an ideal result. (b) An image rendered with one sample per pixel, with no jittering. Note the jaggy artifacts at the edges of checks in the foreground. Notice also the artifacts in the distance where the checker function goes through many cycles between samples; as expected from the signal processing theory presented earlier, that detail reappears incorrectly as lower-frequency aliasing. (c) The result of jittering the image samples, still with just one sample per pixel. The regular aliasing of the second image has been replaced by less objectionable noise artifacts. (d) The result of four jittered samples per pixel is still inferior to the reference image but is substantially better than the previous result.

⟨*StratifiedSampler Declarations*⟩ ≡
```
class StratifiedSampler : public Sampler {
public:
    ⟨StratifiedSampler Public Methods 349⟩
private:
    ⟨StratifiedSampler Private Data 349⟩
};
```

The `StratifiedSampler` generates samples by looping over the pixels from left to right and top to bottom, generating all of the samples for the strata in each pixel before advancing to the next pixel. The constructor takes the range of pixels to generate samples for—`[xstart,ystart]` to `[xend-1,yend-1]`, inclusive—the number of strata in x and y (`xs` and `ys`), a Boolean (`jitter`) that indicates whether the samples should be jittered, and the range of time values where the camera shutter is open, `[sopen, sclose]`.

The implementation of the `GetMoreSamples()` method below generates all `xs * ys` samples for a pixel at once; scratchpad memory that it uses in this process is allocated in the constructor and stored in `sampleBuf`.

⟨*StratifiedSampler Method Definitions*⟩ ≡
```
StratifiedSampler::StratifiedSampler(int xstart, int xend,
        int ystart, int yend, int xs, int ys, bool jitter,
        float sopen, float sclose)
    : Sampler(xstart, xend, ystart, yend, xs * ys, sopen, sclose) {
    jitterSamples = jitter;
    xPos = xPixelStart;
    yPos = yPixelStart;
    xPixelSamples = xs;
    yPixelSamples = ys;
    sampleBuf = new float[5 * xPixelSamples * yPixelSamples];
}
```

The sampler holds the coordinate of the current pixel in the `xPos` and `yPos` member variables, which are initialized to point to the pixel in the upper left of the portion of the image that the sampler is responsible for. In addition to the tiling used for parallelization, the presence of a crop window and implications of sample filtering often cause this corner to start a location other than (0, 0).

Integrator 740
Sampler 340
Sampler::xPixelStart 340
Sampler::yPixelStart 340
StratifiedSampler 349
StratifiedSampler::jitterSamples 349
StratifiedSampler::sampleBuf 349
StratifiedSampler::xPixelSamples 349
StratifiedSampler::xPos 349
StratifiedSampler::yPixelSamples 349
StratifiedSampler::yPos 349

⟨*StratifiedSampler Private Data*⟩ ≡ 349
```
int xPixelSamples, yPixelSamples;
bool jitterSamples;
int xPos, yPos;
float *sampleBuf;
```

The `StratifiedSampler` has no preferred sizes for the number of additional samples generated for the `Integrators`.

⟨*StratifiedSampler Public Methods*⟩ ≡ 349
```
int RoundSize(int size) const { return size; }
```

The GetSubSampler() method uses the ComputeSubwindow() utility routine to compute an image tile extent for the given subsampler number and then returns a new instance of a StratifiedSampler that generates samples in just that region. ComputeSubWindow() may return an empty pixel extent, where one or both of the x and y ranges are degenerate. (Consider, for example, a 2×2 image; it can't be decomposed into more than four image tiles if the tiles are no smaller than a pixel's extent.) In this case, NULL is returned. Calling code must handle this case where a finer decomposition of the image is requested than the Sampler is able to support by ignoring any returned NULL pointers.

⟨*StratifiedSampler Method Definitions*⟩ +≡
```
Sampler *StratifiedSampler::GetSubSampler(int num, int count) {
    int x0, x1, y0, y1;
    ComputeSubWindow(num, count, &x0, &x1, &y0, &y1);
    if (x0 == x1 || y0 == y1) return NULL;
    return new StratifiedSampler(x0, x1, y0, y1, xPixelSamples,
        yPixelSamples, jitterSamples, shutterOpen, shutterClose);
}
```

Direct application of stratification to high-dimensional sampling quickly leads to an intractable number of samples. For example, if we divided the five-dimensional image, lens, and time sample space into four strata in each dimension, the total number of samples per pixel would be $4^5 = 1024$. We could reduce this impact by taking fewer samples in some dimensions (or not stratifying some dimensions, effectively using a single stratum), but we would then lose the benefit of having well-stratified samples in those dimensions. This problem with stratification is known as the *curse of dimensionality*.

We can reap most of the benefits of stratification without paying the price in excessive total sampling by computing lower-dimensional stratified patterns for subsets of the domain's dimensions and then randomly associating samples from each set of dimensions. (This process is sometimes called *padding*.) Figure 7.16 shows the basic idea: we might want to take just four samples per pixel, but still have the samples be stratified over all dimensions. We independently generate four 2D stratified image samples, four 1D stratified time samples, and four 2D stratified lens samples. Then, we randomly associate a time and lens sample value with each image sample. The result is that each pixel has samples that together have good coverage of the sample space. Figure 7.17 shows the improvement in image quality from using stratified lens samples versus using unstratified random samples when rendering depth of field.

Given this approach, it's now possible to implement the GetMoreSamples() method of the StratifiedSampler. It starts by checking whether it is done generating samples for its region—because it generates samples starting from (xPixelStart, yPixelStart) and then first scans across x and then down y, it is done when yPos equals yPixelEnd. If it is done, a return value of zero indicates to the caller that the sampler has completed its work. Otherwise, the method generates a pixel's worth of samples, returning them in the samples array.

Sampler 340
Sampler::
ComputeSubWindow() 341
Sampler::shutterClose 340
Sampler::shutterOpen 340
StratifiedSampler 349
StratifiedSampler::
jitterSamples 349
StratifiedSampler::
xPixelSamples 349
StratifiedSampler::
yPixelSamples 349

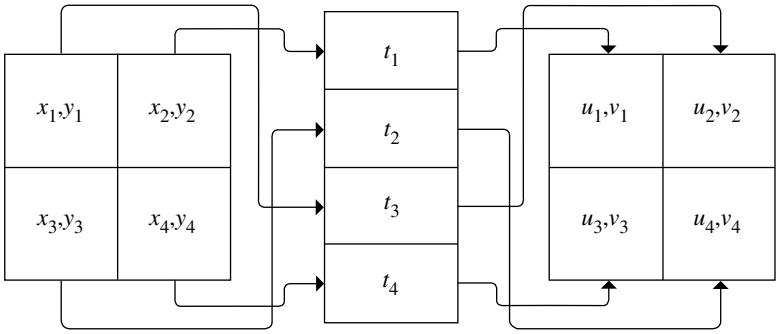


Figure 7.16: We can generate a good sample pattern that reaps the benefits of stratification without requiring that all of the sampling dimensions be stratified simultaneously. Here, we have split (x, y) image position, time t, and (u, v) lens position into independent strata with four regions each. Each is sampled independently, then a time sample and a lens sample are randomly associated with each image sample. We retain the benefits of stratification in each of the individual dimensions without having to exponentially increase the total number of samples.



(a) (b) (c)

Figure 7.17: Effect of Sampling Patterns in Rendering a Red Sphere Image with Depth of Field. (a) A high-quality reference image of the blurred edge of a sphere. (b) An image generated with random sampling in each pixel without stratification. (c) An image generated with the same number of samples, but with the `StratifiedSampler`, which stratified both the image and, more importantly for this image, the lens samples. Stratification makes a substantial improvement for this situation.

⟨*StratifiedSampler Method Definitions*⟩ +≡

```
int StratifiedSampler::GetMoreSamples(Sample *samples, RNG &rng) {
    if (yPos == yPixelEnd) return 0;
    int nSamples = xPixelSamples * yPixelSamples;
    ⟨Generate stratified camera samples for (xPos, yPos) 352⟩
    ⟨Advance to next pixel for stratified sampling 357⟩
    return nSamples;
}
```

RNG 1003
Sample 343
StratifiedSampler 349

⟨*Generate stratified camera samples for* (xPos, yPos)⟩ ≡ 351
⟨*Generate initial stratified samples into* sampleBuf *memory* 352⟩
⟨*Shift stratified image samples to pixel coordinates* 353⟩
⟨*Decorrelate sample dimensions* 353⟩
⟨*Initialize stratified* samples *with sample values* 354⟩

First, stratified sampling utility routines are used to generate two 2D and one 1D sampling patterns for the camera samples. These sample values are stored in the sampleBuf memory for now.

⟨*Generate initial stratified samples into* sampleBuf *memory*⟩ ≡ 352
```
float *bufp = sampleBuf;
float *imageSamples = bufp; bufp += 2 * nSamples;
float *lensSamples = bufp;  bufp += 2 * nSamples;
float *timeSamples = bufp;
StratifiedSample2D(imageSamples, xPixelSamples, yPixelSamples, rng,
                   jitterSamples);
StratifiedSample2D(lensSamples, xPixelSamples, yPixelSamples, rng,
                   jitterSamples);
StratifiedSample1D(timeSamples, xPixelSamples * yPixelSamples, rng,
                   jitterSamples);
```

The 1D and 2D stratified sampling routines are implemented as utility functions, since they will be useful elsewhere in pbrt. Both of them loop over the given number of strata over the [0, 1] domain and place a sample value in each one.

⟨*Sampling Function Definitions*⟩ ≡
```
void StratifiedSample1D(float *samp, int nSamples, RNG &rng,
                        bool jitter) {
    float invTot = 1.f / nSamples;
    for (int i = 0;  i < nSamples; ++i) {
        float delta = jitter ? rng.RandomFloat() : 0.5f;
        *samp++ = (i + delta) * invTot;
    }
}
```

⟨*Sampling Function Definitions*⟩ +≡
```
void StratifiedSample2D(float *samp, int nx, int ny, RNG &rng,
                        bool jitter) {
    float dx = 1.f / nx, dy = 1.f / ny;
    for (int y = 0; y < ny; ++y)
        for (int x = 0; x < nx; ++x) {
            float jx = jitter ? rng.RandomFloat() : 0.5f;
            float jy = jitter ? rng.RandomFloat() : 0.5f;
            *samp++ = (x + jx) * dx;
            *samp++ = (y + jy) * dy;
        }
}
```

RNG 1003
RNG::RandomFloat() 1003
StratifiedSample1D() 352
StratifiedSample2D() 352
StratifiedSampler::
jitterSamples 349
StratifiedSampler::
sampleBuf 349
StratifiedSampler::
xPixelSamples 349
StratifiedSampler::
yPixelSamples 349

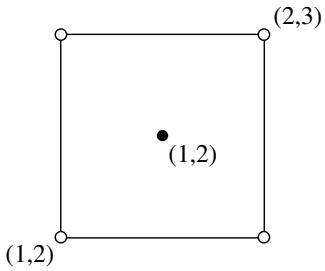


Figure 7.18: To generate stratified samples for a discrete pixel $(1, 2)$, all that needs to be done is to add $(1, 2)$ to the elements of the samples generated over $[0, 1)^2$. This gives samples with continuous pixel coordinates from $(1, 2)$ to $(2, 3)$, which end up surrounding the discrete pixel $(1, 2)$.

The `StratifiedSample2D()` utility function generates samples in the range $[0, 1)^2$, but image samples need to be expressed in terms of continuous pixel coordinates. Therefore, `GetMoreSamples()` next loops over all of the new stratified samples and adds the current (x, y) pixel number, so that the samples for the discrete pixel (x, y) range over the continuous coordinates $[x, x + 1) \times [y, y + 1)$, following the convention for continuous pixel coordinates described in Section 7.1.7. Figure 7.18 reviews the relationship between discrete and continuous coordinates as it relates to samples for a pixel.

⟨*Shift stratified image samples to pixel coordinates*⟩ ≡ 352
```
for (int o = 0; o < 2 * xPixelSamples * yPixelSamples; o += 2) {
    imageSamples[o]   += xPos;
    imageSamples[o+1] += yPos;
}
```

In order to randomly associate a time and lens sample with each image sample, this method next shuffles the order of the time and lens sample arrays. Thus, when it later initializes a `Sample` with the `i`th precomputed sample value for this pixel, it can just return the `i`th time and lens sample.

`Shuffle()` 354
`StratifiedSample2D()` 352
`StratifiedSampler::xPixelSamples` 349
`StratifiedSampler::xPos` 349
`StratifiedSampler::yPixelSamples` 349
`StratifiedSampler::yPos` 349

⟨*Decorrelate sample dimensions*⟩ ≡ 352
```
Shuffle(lensSamples, xPixelSamples*yPixelSamples, 2, rng);
Shuffle(timeSamples, xPixelSamples*yPixelSamples, 1, rng);
```

The `Shuffle()` utility function randomly permutes a sample pattern of `count` samples in `dims` dimensions.

⟨*Monte Carlo Utility Declarations*⟩ ≡
```
template <typename T>
void Shuffle(T *samp, uint32_t count, uint32_t dims, RNG &rng) {
    for (uint32_t i = 0; i < count; ++i) {
        uint32_t other = i + (rng.RandomUInt() % (count - i));
        for (uint32_t j = 0; j < dims; ++j)
            swap(samp[dims*i + j], samp[dims*other + j]);
    }
}
```

Now that the camera samples have been offset and the lens and time samples have been shuffled, the sample values can be copied out of the temporary buffer and stored in the array of `samples` passed to `GetMoreSamples()`. As this step is performed, sample values for the samples requested by the integrators are generated for each of the returned samples as well.

⟨*Initialize stratified* `samples` *with sample values*⟩ ≡ 352
```
for (int i = 0; i < nSamples; ++i) {
    samples[i].imageX = imageSamples[2*i];
    samples[i].imageY = imageSamples[2*i+1];
    samples[i].lensU = lensSamples[2*i];
    samples[i].lensV = lensSamples[2*i+1];
    samples[i].time = Lerp(timeSamples[i], shutterOpen, shutterClose);
    ⟨Generate stratified samples for integrators 356⟩
}
```

Integrators introduce a new complication since they often use multiple samples per image sample in some dimensions rather than a single sample value like the camera does for lens position and time. As we have described the topic of sampling so far, this leaves us with a quandary: if an integrator asks for a set of 64 two-dimensional sample values for each image sample, the sampler has two different goals to try to fulfill:

1. We would like each image sample's 64 integrator samples to themselves be well distributed in 2D (i.e., with an 8×8 stratified grid). Stratification here will improve the quality of the integrator's results for each individual sample.
2. We would like to ensure that the set of integrator samples for one image sample isn't too similar to the samples for its neighboring image samples. As with time and lens samples, we'd like the points to be well distributed with respect to their neighbors, so that over the region around a single pixel, there is good coverage of the entire sample space.

Rather than trying to solve both of these problems simultaneously here, the `StratifiedSampler` will only address the first one. The other samplers later in this chapter will revisit this issue with more sophisticated techniques and solve both of them simultaneously to various degrees.

A second integrator-related complication comes from the fact that they may ask for an arbitrary number of samples per image sample, so stratification may not be easily applied. (For example, how do we generate a stratified 2D pattern of seven samples?) We could just generate an $n \times 1$ or $1 \times n$ stratified pattern, but this only gives us the

CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
Lerp() 1000
RNG 1003
RNG::RandomUInt() 1003
Sampler::shutterClose 340
Sampler::shutterOpen 340
StratifiedSampler 349

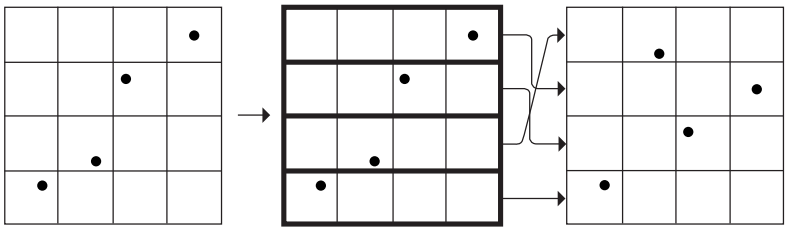


Figure 7.19: Latin hypercube sampling (sometimes called n-rooks sampling) chooses samples such that only a single sample is present in each row and each column of a grid. This can be done by generating random samples in the cells along the diagonal and then randomly permuting their coordinates. One advantage of LHS is that it can generate any number of samples with a good distribution, not just $m \times n$ samples, as with stratified patterns.

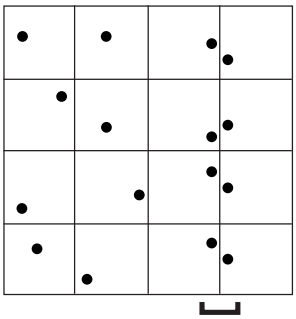


Figure 7.20: A Worst-Case Situation for Stratified Sampling. In an $n \times n$ 2D pattern, up to $2n$ of the points may project to essentially the same point on one of the axes. When "unlucky" patterns like this are generated, the quality of the results computed with them usually suffers.

benefit of stratification in one dimension, and no guarantee of a good pattern in the other dimension. The `StratifiedSampler::RoundSize()` method could round requests up to the next number that's the square of integers, but instead we will use an approach called *Latin hypercube sampling* (LHS), which can generate any number of samples in any number of dimensions with reasonably good distribution.

LHS uniformly divides each dimension's axis into n regions and generates a jittered sample in each of the n regions along the diagonal, as shown on the left in Figure 7.19. These samples are then randomly shuffled in each dimension, creating a pattern with good distribution. An advantage of LHS is that it minimizes clumping of the samples when they are projected onto any of the axes of the sampling dimensions. This is in contrast to stratified sampling, where $2n$ of the $n \times n$ samples in a 2D pattern may project to essentially the same point on each of the axes. Figure 7.20 shows this worst-case situation for a stratified sampling pattern.

StratifiedSampler::
RoundSize() 349

In spite of addressing the clumping problem, LHS isn't necessarily an improvement to stratified sampling; it's easy to construct cases where the sample positions are essentially colinear and large areas of $[0, 1]^2$ have no samples near them (e.g., when the permutation of the original samples is the identity, leaving them all where they started). In particular, as n increases, Latin hypercube patterns are less and less effective compared to stratified patterns. We will revisit this issue in the next section, where we will discuss sample patterns that are simultaneously stratified and distributed in a Latin hypercube pattern.

⟨*Generate stratified samples for integrators*⟩ ≡ 354
```
for (uint32_t j = 0; j < samples[i].n1D.size(); ++j)
    LatinHypercube(samples[i].oneD[j], samples[i].n1D[j], 1, rng);
for (uint32_t j = 0; j < samples[i].n2D.size(); ++j)
    LatinHypercube(samples[i].twoD[j], samples[i].n2D[j], 2, rng);
```

The general-purpose `LatinHypercube()` function generates an arbitrary number of LHS samples in an arbitrary dimension. The number of elements in the `samples` array should thus be `nSamples*nDim`.

⟨*Sampling Function Definitions*⟩ +≡
```
void LatinHypercube(float *samples, uint32_t nSamples, uint32_t nDim,
                    RNG &rng) {
    ⟨Generate LHS samples along diagonal 356⟩
    ⟨Permute LHS samples in each dimension 356⟩
}
```

⟨*Generate LHS samples along diagonal*⟩ ≡ 356
```
float delta = 1.f / nSamples;
for (uint32_t i = 0; i < nSamples; ++i)
    for (uint32_t j = 0; j < nDim; ++j)
        samples[nDim * i + j] = (i + (rng.RandomFloat())) * delta;
```

To do the permutation, this function loops over the samples, randomly permuting the sample points in one dimension at a time. Note that this is a different permutation than the earlier `Shuffle()` routine: that routine does one permutation, keeping all `nDim` sample points in each sample together, while here `nDim` separate permutations of a single dimension at a time are done (Figure 7.21).[4]

⟨*Permute LHS samples in each dimension*⟩ ≡ 356
```
for (uint32_t i = 0; i < nDim; ++i) {
    for (uint32_t j = 0; j < nSamples; ++j) {
        uint32_t other = j + (rng.RandomUInt() % (nSamples - j));
        swap(samples[nDim * j + i], samples[nDim * other + i]);
    }
}
```

LatinHypercube() 356
RNG 1003
RNG::RandomFloat() 1003
RNG::RandomUInt() 1003
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
Shuffle() 354

4 While it's not necessary to permute the first dimension of the LHS pattern, the implementation here does so anyway, since making the elements of the first dimension be randomly ordered means that LHS patterns can be used in conjunction with sampling patterns from other sources without danger of correlation between their sample points.

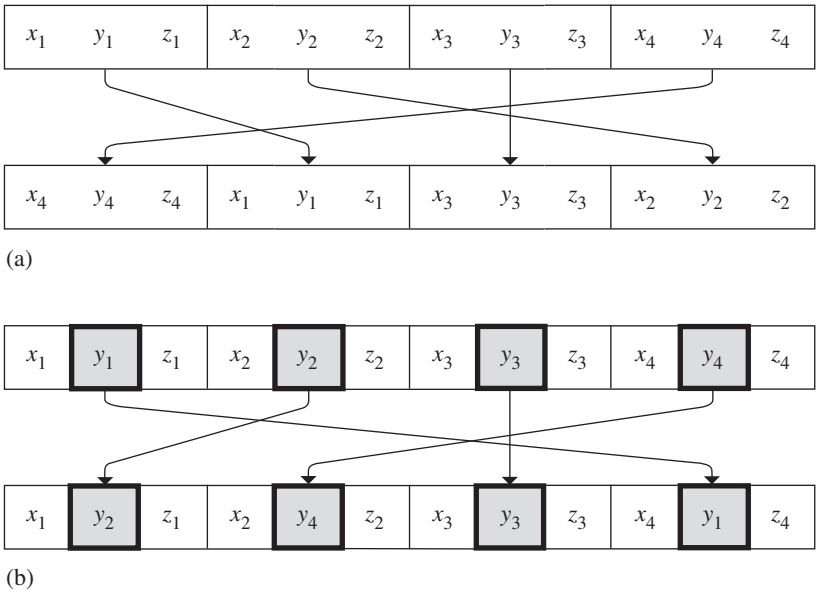


Figure 7.21: (a) The permutation done by the `Shuffle()` routine moves entire blocks of `nDims` elements around. (b) The permutation for Latin hypercube sampling permutes each dimension's samples independently. Here, the shuffling of the second dimension's samples from a four-element pattern of three dimensions is shown.

Starting with the scene in Figure 7.22, Figure 7.23 shows the improvement from good samples for the `DirectLightingIntegrator`. Image (a) was computed with 1 image sample per pixel, each with 16 shadow samples, and image (b) was computed with 16 image samples per pixel, each with 1 shadow sample. Because the `StratifiedSampler` could generate a good LHS pattern for the first case, the quality of the shadow is much better, even with the same total number of shadow samples taken.

After a pixel's worth of samples have been generated, the `StratifiedSampler` advances the pixel location to the next pixel. It first tries to move one pixel over in the x direction. If doing so takes it beyond the end of the sampling range, it resets the x position to the first pixel in the next x row of pixels and advances the y position. Because the y pixel location `yPos` is advanced only when the end of a row of pixels in the x direction is reached, once the y position counter has advanced past the bottom of the sampling range, sample generation is complete.

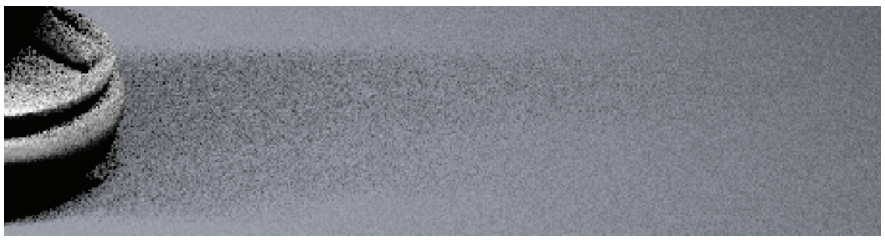
DirectLightingIntegrator 742
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Shuffle() 354
StratifiedSampler 349
StratifiedSampler::xPos 349
StratifiedSampler::yPos 349

⟨*Advance to next pixel for stratified sampling*⟩ ≡ 351

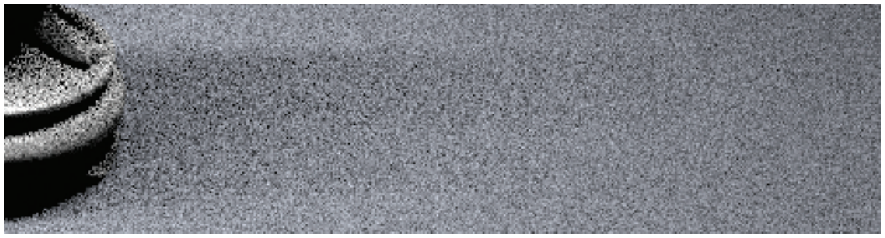
```
if (++xPos == xPixelEnd) {
    xPos = xPixelStart;
    ++yPos;
}
```



Figure 7.22: Area Light Sampling Example Scene.



(a)



(b)

Figure 7.23: Sampling an Area Light with Samples from the Stratified Sampler. (a) shows the result of using 1 image sample per pixel and 16 shadow samples, and (b) shows the result of 16 image samples, each with just 1 shadow sample. The total number of shadow samples is the same in both cases, but because the version with 16 shadow samples per image sample is able to use an LHS pattern, all of the shadow samples in a pixel's area are well distributed, while in the second image the implementation here has no way to prevent them from being poorly distributed. The difference is striking.

Finally, the `MaximumSampleCount()` method can be implemented; recall that each call to `GetMoreSamples()` returns `xPixelSamples * yPixelSamples` until sampling is done.

⟨*StratifiedSampler Public Methods*⟩ +≡ 349
```
int MaximumSampleCount() { return xPixelSamples * yPixelSamples; }
```

*7.4 LOW-DISCREPANCY SAMPLING

The underlying goal of the `StratifiedSampler` is to generate a well-distributed but not uniform set of sample points, with no two sample points too close together and no excessively large regions of the sample space that have no samples. As Figure 7.14 showed, a jittered pattern does this much better than a random pattern does, although its quality can suffer when samples in adjacent strata happen to be close to the shared boundary of their two strata.

Mathematicians have developed a concept called *discrepancy* that can be used to evaluate the quality of a pattern of sample positions like these in a way that their quality can be expressed numerically. Patterns that are well distributed (in a manner to be formalized shortly) have low-discrepancy values, and thus the sample pattern generation problem can be considered to be one of finding a suitable *low-discrepancy* pattern of points.[5] A number of deterministic techniques have been developed that generate low-discrepancy point sets, even in high-dimensional spaces. This section will use a few of them as the basis for a low-discrepancy sample generator.

7.4.1 DEFINITION OF DISCREPANCY

Before defining the low-discrepancy sampling class `LDSampler`, we will first introduce a formal definition of discrepancy. The basic idea is that the "quality" of a set of points in an n-dimensional space $[0, 1]^n$ can be evaluated by looking at regions of the domain $[0, 1]^n$, counting the number of points inside each region, and comparing the volume of each region to the number of sample points inside. In general, a given fraction of the volume should have roughly the same fraction of the sample points inside of it. While it's not possible for this always to be the case, we can still try to use patterns that minimize the difference between the actual volume and the volume estimated by the points (the *discrepancy*). Figure 7.24 shows an example of the idea in two dimensions.

To compute the discrepancy of a set of points, we first pick a family of shapes B that are subsets of $[0, 1]^n$. For example, boxes with one corner at the origin are often used. This corresponds to

$$B = \{[0, v_1] \times [0, v_2] \times \cdots \times [0, v_s]\},$$

where $0 \leq v_i \leq 1$. Given a sequence of sample points $P = x_1, \ldots, x_N$, the discrepancy

LDSampler 373
StratifiedSampler 349
StratifiedSampler::
xPixelSamples 349
StratifiedSampler::
yPixelSamples 349

5 Of course, using discrepancy in this way implicitly assumes that the metric used to compute discrepancy is one that has good correlation with the quality of a pattern for image sampling, which may be a slightly different thing, particularly given the involvement of the human visual system in the process.

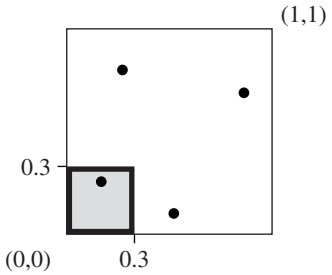


Figure 7.24: The discrepancy of a box (shaded) given a set of 2D sample points in $[0, 1]^2$. One of the four sample points is inside the box, so this set of points would estimate the box's area to be $1/4$. The true area of the box is $.3 \times .3 = .09$, so the discrepancy for this particular box is $.25 - .09 = .16$. In general, we're interested in finding the maximum discrepancy of all possible boxes (or some other shape) to compute discrepancy.

of P with respect to B is[6]

$$D_N(B, P) = \sup_{b \in B} \left| \frac{\sharp\{x_i \in b\}}{N} - \lambda(b) \right|,$$

where $\sharp\{x_i \in b\}$ is the number of points in b and $\lambda(b)$ is the volume of b.

The intuition for why this is a reasonable measure of quality is that $\sharp\{x_i \in b\}/N$ is an approximation of the volume of the box b given by the particular points P. Therefore, the discrepancy is the worst error over all possible boxes from this way of approximating volume. When the set of shapes B is the set of boxes with a corner at the origin, this is called the *star discrepancy*, $D_N^*(P)$. Another popular option for B is the set of all axis-aligned boxes, where the restriction that one corner be at the origin has been removed.

For a few particular point sets, the discrepancy can be computed analytically. For example, consider the set of points in one dimension

$$x_i = \frac{i}{N}.$$

We can see that the star discrepancy of x_i is

$$D_N^*(x_1, \ldots, x_n) = \frac{1}{N}.$$

For example, take the interval $b = [0, 1/N)$. Then $\lambda(b) = 1/N$, but $\sharp\{x_i \in b\} = 0$. This interval (and the intervals $[0, 2/N)$, etc.) is the interval where the largest differences between volume and fraction of points inside the volume are seen.

The star discrepancy of this sequence can be improved by modifying it slightly:

6 The sup operator is the continuous analog of the discrete max operator. That is, sup $f(x)$ is a constant-valued function of x that passes through the maximum value taken on by $f(x)$.

$$x_i = \frac{i - \frac{1}{2}}{N}.$$

Then

$$D_N^*(x_i) = \frac{1}{2N}.$$

The bounds for the star discrepancy of a sequence of points in one dimension have been shown to be

$$D_N^*(x_i) = \frac{1}{2N} + \max_{1 \le i \le N} \left| x_i - \frac{2i - 1}{2N} \right|.$$

Thus, the earlier modified sequence has the lowest possible discrepancy for a sequence in 1D. In general, it is much easier to analyze and compute bounds for the discrepancy of sequences in 1D than for those in higher dimensions. For less simply constructed point sequences, and for sequences in higher dimensions and for more irregular shapes than boxes, the discrepancy often must be estimated numerically by constructing a large number of shapes B, computing their discrepancy, and reporting the maximum.

The astute reader will notice that according to the low-discrepancy measure, this uniform sequence in 1D is optimal, but earlier in this chapter we claimed that irregular jittered patterns were perceptually superior to uniform patterns for image sampling in 2D since they replaced aliasing error with noise. In that framework, uniform samples are clearly not optimal. Fortunately, low-discrepancy patterns in higher dimensions are much less uniform than they are in one dimension and thus usually work reasonably well as sample patterns in practice. Nevertheless, their underlying uniformity is probably the reason why low-discrepancy patterns can be more prone to aliasing than patterns with true pseudo-random variation.

7.4.2 HAMMERSLEY AND HALTON SEQUENCES

We will now introduce a number of techniques that have been developed specifically to generate sequences of points that have low discrepancy. Remarkably, few lines of code are necessary to compute many low-discrepancy sampling patterns.

The first set of techniques that we will describe use a construction called the *radical inverse*. It is based on the fact that a positive integer value n can be expressed in a base b with a sequence of digits $d_m \ldots d_2 d_1$ uniquely determined by

$$n = \sum_{i=1}^{\infty} d_i b^{i-1}.$$

The radical inverse function Φ_b in base b converts a nonnegative integer n to a floating-point value in $[0, 1)$ by reflecting these digits about the decimal point:

$$\Phi_b(n) = 0.d_1 d_2 \ldots d_m.$$

Thus, the contribution of the digit d_i to the radical inverse is d_i / b^i.

RadicalInverse() 362

The function `RadicalInverse()` computes the radical inverse for a given number `n` in the base `base`. It first computes the value of d_1 by taking the remainder of the number n when divided by the base and adds $d_1 b^{-1}$ to the radical inverse value. It then divides n by the

base, effectively chopping off the last digit, so that the next time through the loop it can compute d_2 by finding the remainder in base `base` and adding $d_2 b^{-2}$ to the sum, and so on. This process continues until n is zero, at which point it has found the last nonzero d_i value.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline double RadicalInverse(int n, int base) {
    double val = 0;
    double invBase = 1. / base, invBi = invBase;
    while (n > 0) {
        ⟨Compute next digit of radical inverse 362⟩
    }
    return val;
}
```

⟨*Compute next digit of radical inverse*⟩ ≡ 362
```
int d_i = (n % base);
val += d_i * invBi;
n *= invBase;
invBi *= invBase;
```

The `RadicalInverse()` function can be implemented much more efficiently for base 2, with bit operations replacing the floating-point math above. Furthermore, for the common case where the value `n` is incremented by one for each successive call to the radical inverse function, it's possible to incrementally generate each successive point more efficiently than computing its value from scratch. Because the `RadicalInverse()` function isn't currently used in performance-critical code in `pbrt`, we leave these improvements for later work.

One of the simplest low-discrepancy sequences is the van der Corput sequence, which is a one-dimensional sequence given by the radical inverse function in base 2:

$$x_i = \Phi_2(i).$$

Table 7.2 shows the first few values of the van der Corput sequence. Notice how it recursively splits the intervals of the 1D line in half, generating a sample point at the center of each interval. The discrepancy of this sequence is

$$D_N^*(P) = O\left(\frac{\log N}{N}\right),$$

which matches the best discrepancy that has been attained for infinite sequences in d dimensions,

$$D_N^*(P) = O\left(\frac{(\log N)^d}{N}\right).$$

Two well-known low-discrepancy sequences that are defined in an arbitrary number of dimensions are the *Halton* and *Hammersley* sequences. Both use the radical inverse function as well. To generate an n-dimensional Halton sequence, we use the radical

Table 7.2: The radical inverse $\Phi_2(n)$ of the first few positive integers, computed in base 2. Notice how successive values of $\Phi_2(n)$ are not close to any of the previous values of $\Phi_2(n)$. As more and more values of the sequence are generated, samples are necessarily closer to previous samples, although with a minimum distance that is guaranteed to be reasonably good.

n	**Base 2**	$\Phi_2(n)$
1	1	$.1 = 1/2$
2	10	$.01 = 1/4$
3	11	$.11 = 3/4$
4	100	$.001 = 1/8$
5	101	$.101 = 5/8$
⋮	⋮	⋮

inverse base b, with a different base for each dimension of the pattern. The bases used must all be relatively prime to each other, so a natural choice is to use the first n prime numbers $(p_1, \ldots, p_n)$:

$$x_i = (\Phi_2(i), \Phi_3(i), \Phi_5(i), \ldots, \Phi_{p_n}(i)).$$

One of the most useful characteristics of the Halton sequence is that it can be used even if the total number of samples needed isn't known in advance; all prefixes of the sequence are well distributed, so as additional samples are added to the sequence low discrepancy will be maintained. This property will be used by the `PhotonIntegrator`, for example, which doesn't know the total number of photons that will be necessary to emit from lights in the scene ahead of time and thus uses a Halton sequence to get a well-distributed set of photons.

The discrepancy of a d-dimensional Halton sequence is

$$D_N^*(x_i) = O\left(\frac{(\log N)^d}{N}\right),$$

which is asymptotically optimal.

If the number of samples N is fixed, the Hammersley point set can be used, giving slightly lower discrepancy. Hammersley point sets are defined by

$$x_i = \left(\frac{i}{N}, \Phi_{b_1}(i), \Phi_{b_2}(i), \ldots, \Phi_{b_n}(i)\right),$$

where N is the total number of samples to be taken and as before all of the bases b_i are relatively prime. Figure 7.25(a) shows a plot of the first 100 points of the 2D Halton sequence. Figure 7.25(b) shows the Hammersley sequence.

The Halton Sampler

LDSampler 373

PhotonIntegrator 802

The `HaltonSampler` generates image, lens, and time samples using the Halton sequence. For each camera sample, it generates samples for the integrators using a Latin hypercube pattern. This sampler isn't as effective as the `LDSampler` that will be introduced shortly,

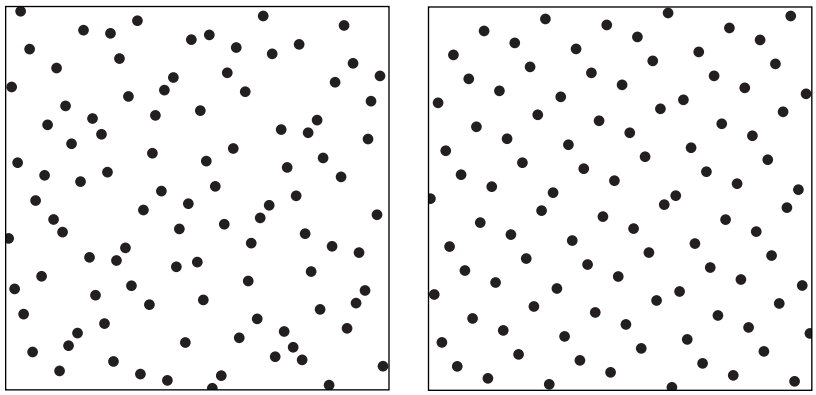


Figure 7.25: The First 100 Points of Two Low-Discrepancy Sequences in 2D. (a) Halton, (b) Hammersley.

but its implementation is straightforward and it generates reasonably good sampling patterns. However, it can be prone to aliasing in images. Figure 7.26 compares the results of sampling a checkerboard texture using a Halton-based sampler to using the stratified sampler from the previous section. Note the unpleasant pattern along edges in the foreground and toward the horizon.

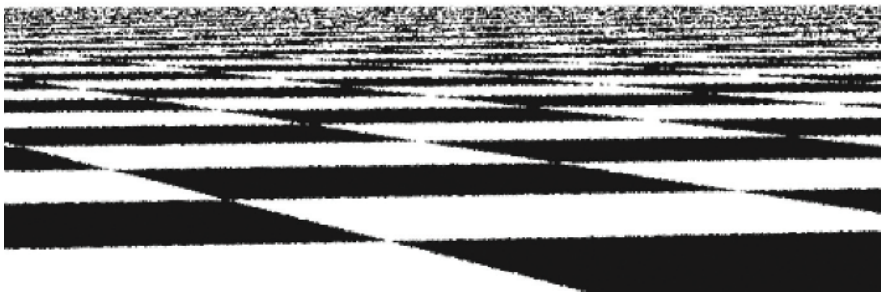
The `HaltonSampler` constructor takes the usual parameters that give the pixel extent to sample, the number of samples to take per pixel, and the shutter open range. It then computes the total number of samples it will generate, `wantedSamples`, and initializes `currentSample` to zero.

⟨*HaltonSampler Private Data*⟩ ≡
```
int wantedSamples, currentSample;
```

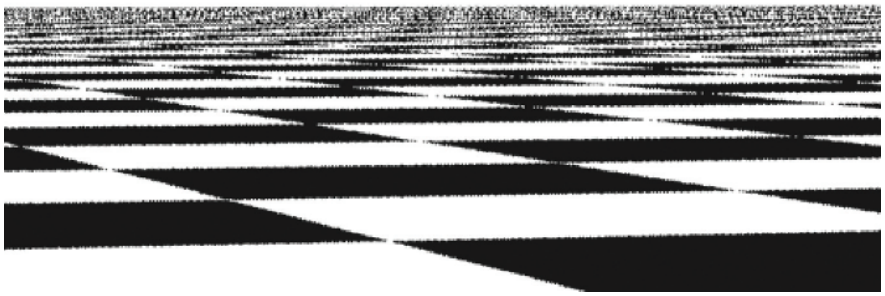
The only subtlety in the implementation is how it handles nonsquare images. The implementation here computes a total number of samples to take as if it were generating samples for a square image with resolution given by the greater of the x and y resolutions. It then ignores samples that are outside the image extent.

⟨*HaltonSampler Method Definitions*⟩ ≡
```
int HaltonSampler::GetMoreSamples(Sample *samples, RNG &rng) {
retry:
    if (currentSample >= wantedSamples) return 0;
    ⟨Generate sample with Halton sequence and reject if outside image extent 365⟩
    ⟨Generate lens, time, and integrator samples for HaltonSampler 366⟩
    return 1;
}
```

HaltonSampler::
currentSample 364
HaltonSampler::
wantedSamples 364
RNG 1003
Sample 343



(a)



(b)

Figure 7.26: Comparison of the Stratified Sampler to a Low-Discrepancy Sampler Based on Halton Points on the Image Plane. (a) The jittered stratified sampler with a single sample per pixel and (b) the `HaltonSampler` sampler with a single sample per pixel. Note that although the Halton pattern is able to reproduce the checker pattern farther toward the horizon than the stratified pattern, there is a regular structure to the error in the low-discrepancy pattern that is visually distracting; it doesn't turn aliasing into less objectionable noise as well as the jittered approach. (With one sample per pixel, the `HaltonSampler` performs similarly to the jittered sampler.)

The first thing `GetMoreSamples()` does is generate the `imageX` and `imageY` samples. If these are outside of the sampling extent, they are rejected and we try again with the next sample number.

⟨*Generate sample with Halton sequence and reject if outside image extent*⟩ ≡ 364

```
float u = (float)RadicalInverse(currentSample, 3);
float v = (float)RadicalInverse(currentSample, 2);
float lerpDelta = float(max(xPixelEnd - xPixelStart,
                            yPixelEnd - yPixelStart));
samples->imageX = Lerp(u, xPixelStart, xPixelStart + lerpDelta);
samples->imageY = Lerp(v, yPixelStart, yPixelStart + lerpDelta);
++currentSample;
if (samples->imageX >= xPixelEnd || samples->imageY >= yPixelEnd)
    goto retry;
```

CameraSample::imageX 342
CameraSample::imageY 342
HaltonSampler 364
HaltonSampler::
currentSample 364
RadicalInverse() 362
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340

If the image sample is within the image extent, then samples for the remaining dimensions are generated.

⟨*Generate lens, time, and integrator samples for* HaltonSampler⟩ ≡ 364
```
samples->lensU = (float)RadicalInverse(currentSample, 5);
samples->lensV = (float)RadicalInverse(currentSample, 7);
samples->time = Lerp((float)RadicalInverse(currentSample, 11),
                     shutterOpen, shutterClose);
for (uint32_t i = 0; i < samples->n1D.size(); ++i)
    LatinHypercube(samples->oneD[i], samples->n1D[i], 1, rng);
for (uint32_t i = 0; i < samples->n2D.size(); ++i)
    LatinHypercube(samples->twoD[i], samples->n2D[i], 2, rng);
```

Randomized Halton Sequences

The Hammersley and Halton sequences have a few shortcomings. First, while their first few dimensions are irregularly distributed in desirable ways, as the base b increases, the values can exhibit surprisingly regular patterns. Second, these sequences aren't well suited to parallel rendering. If multiple threads of execution are all independently generating samples from a Halton or Hammersley sequence, they will all generate the exact same set of samples, likely computing the exact same results, which is probably not desirable. On the other hand, the synchronization overhead of having all threads draw samples from a single sequence would likely be unacceptable.

An alternative approach is to use *randomized Halton sequences*, where each instance of the sequence is independent from the others. A variety of approaches have been suggested for doing this, but a simple and effective one is to compute a permutation table for the digits and use it when computing the radical inverse:

$$\Psi_b(n) = 0.p(d_1)p(d_2)\ldots p(d_m),$$

where p is a random permutation of the digits $(0, 1, \ldots, b-1)$. Note that the same permutation is used for each digit, and the same permutation is used for generating each of the n sample points. `GeneratePermutation()` is a simple utility function to generate such a permutation for base b.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline void GeneratePermutation(uint32_t *buf, uint32_t b, RNG &rng) {
    for (uint32_t i = 0; i < b; ++i)
        buf[i] = i;
    Shuffle(buf, b, 1, rng);
}
```

`PermutedRadicalInverse()` is a modified version of `RadicalInverse()` that applies the given permutation to each digit.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline double PermutedRadicalInverse(uint32_t n, uint32_t base,
                                     const uint32_t *p) {
    double val = 0;
    double invBase = 1. / base, invBi = invBase;
```

CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
HaltonSampler::currentSample 364
LatinHypercube() 356
RadicalInverse() 362
RNG 1003
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
Sampler::shutterClose 340
Sampler::shutterOpen 340
Shuffle() 354

```
        while (n > 0) {
            uint32_t d_i = p[n % base];
            val += d_i * invBi;
            n *= invBase;
            invBi *= invBase;
        }
        return val;
    }
```

`PermutedHalton` is a utility class that generates a sequence of sample points in a given dimension using randomly permuted Halton values.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
    class PermutedHalton {
    public:
        ⟨PermutedHalton Public Methods 368⟩
    private:
        ⟨PermutedHalton Private Data 368⟩
    };
```

Its constructor takes the dimensionality of sample points to generate, `d`. It uses a precomputed array of prime numbers, `primes`, to determine the base b_i for each dimension and computes a permutation table for each one.

⟨*Monte Carlo Function Definitions*⟩ ≡
```
    PermutedHalton::PermutedHalton(uint32_t d, RNG &rng) {
        dims = d;
        ⟨Determine bases b_i and their sum 367⟩
        ⟨Compute permutation tables for each base 368⟩
    }
```

Recall that each dimension requires a permutation table of size b_i; thus, the total number of entries that must be stored for all of the permutation tables is given by the sum of the bases b_i—this value is computed here as the base for each dimension is set.

⟨*Determine bases b_i and their sum*⟩ ≡ 367
```
    b = new uint32_t[dims];
    uint32_t sumBases = 0;
    for (uint32_t i = 0; i < dims; ++i) {
        b[i] = primes[i];
        sumBases += b[i];
    }
```

PermutedHalton 367
PermutedHalton::b 368
PermutedHalton::dims 368
primes 368
RNG 1003

Memory for all of the permutation tables is allocated in a single allocation that's large enough to hold all of the entries. The first `b[0]` entries hold the permutation table for the first dimension, the next `b[1]` entries hold the table for the second dimension, and so forth.

⟨*Compute permutation tables for each base*⟩ ≡ 367
```
permute = new uint32_t[sumBases];
uint32_t *p = permute;
for (uint32_t i = 0; i < dims; ++i) {
    GeneratePermutation(p, b[i], rng);
    p += b[i];
}
```

⟨*PermutedHalton Private Data*⟩ ≡ 367
```
uint32_t dims;
uint32_t *b, *permute;
```

⟨*Sampling Local Definitions*⟩ ≡
```
static const int primes[] = {
    ⟨First 1000 prime numbers⟩
};
```

Given all this, the `PermutedHalton`'s `Sample()` method generates the `n`th sample of the sequence, storing the `dims` values in the `out` array.

⟨*PermutedHalton Public Methods*⟩ ≡ 367
```
void Sample(uint32_t n, float *out) const {
    uint32_t *p = permute;
    for (uint32_t i = 0; i < dims; ++i) {
        out[i] = PermutedRadicalInverse(n, b[i], p);
        p += b[i];
    }
}
```

7.4.3 (0,2)-SEQUENCES

To generate high-quality samples for the integrators, we can take advantage of a remarkable property of certain low-discrepancy patterns that allows us to satisfy both parts of our original goal for samplers (only one of which was satisfied with the `StratifiedSampler`): they make it possible to generate a set of integrator samples for a pixel's worth of image samples such that the samples for each pixel sample are well distributed with respect to each other, and such that the aggregate collection of integrator samples for all of the pixel samples in the pixel are collectively well distributed.

A useful low-discrepancy sequence in 2D can be constructed using the van der Corput sequence in one dimension and a sequence based on a radical inverse function due to Sobol′ in the other direction. The resulting sequence is a special type of low-discrepancy sequence known as an (0, 2)-sequence. (0, 2)-sequences are stratified in a very general way. For example, the first 16 samples in an (0, 2)-sequence satisfy the stratification constraint from Section 7.3, meaning there is just one sample in each of the boxes of extent $(\frac{1}{4}, \frac{1}{4})$. However, they also satisfy the Latin hypercube constraint, as only one of them is in each of the boxes of extent $(\frac{1}{16}, 1)$ and $(1, \frac{1}{16})$. Furthermore, there is only one sample in each of the boxes of extent $(\frac{1}{2}, \frac{1}{8})$ and $(\frac{1}{8}, \frac{1}{2})$. Figure 7.27 shows all of the possibilities for dividing the domain into regions where the first 16 samples of

GeneratePermutation() 366
PermutedHalton::b 368
PermutedHalton::dims 368
PermutedHalton::permute 368
PermutedRadicalInverse() 366
StratifiedSampler 349

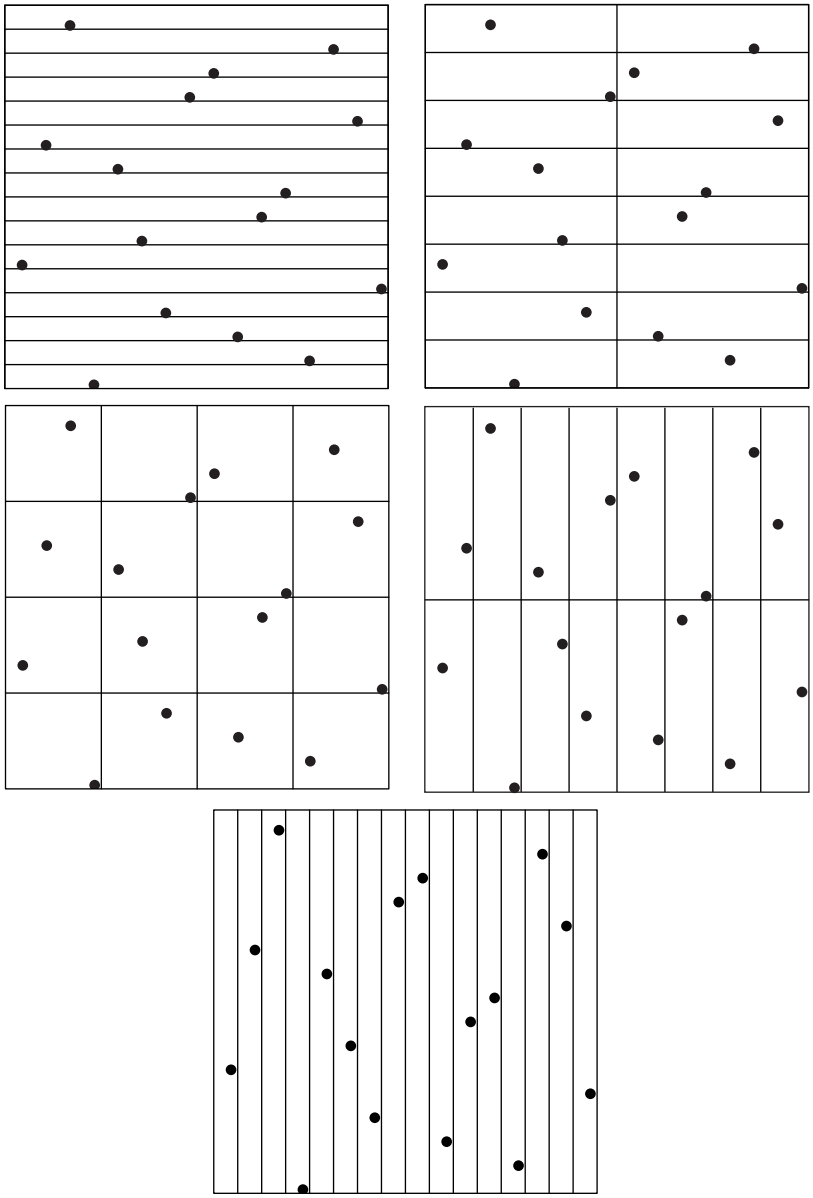


Figure 7.27: A sampling pattern that has a single sample in all of the base 2 elementary intervals. It satisfies both the 4×4 stratification and Latin hypercube constraints as well as the other stratification constraints shown.

an (0, 2)-sequence satisfy the stratification properties. Each succeeding sequence of 16 samples from this pattern also satisfies the distribution properties.

In general, any sequence of length $2^{l_1+l_2}$ (where l_i is a nonnegative integer) from an (0, 2)-sequence satisfies this general stratification constraint. The set of *elementary intervals* in two dimensions, base 2, is defined as

$$E = \left\{ \left[\frac{a_1}{2^{l_1}}, \frac{a_1+1}{2^{l_1}} \right) \times \left[\frac{a_2}{2^{l_2}}, \frac{a_2+1}{2^{l_2}} \right) \right\},$$

where the integer $a_i = 0, \ldots, 2^{l_i} - 1$. One sample from each of the first $2^{l_1+l_2}$ values in the sequence will be in each of the elementary intervals. Furthermore, the same property is true for each subsequent set of $2^{l_1+l_2}$ values.

To understand now how (0, 2)-sequences can be applied to generating 2D samples for the integrators, consider a pixel with 2×2 image samples, each with 4×4 integrator samples. The first $2 \times 2 \times 4 \times 4 = 2^6$ values of an (0, 2)-sequence are well distributed with respect to each other according to the corresponding set of elementary intervals. Furthermore, the first $4 \times 4 = 2^4$ samples are themselves well distributed according to their corresponding elementary intervals, as are the next 2^4 of them, and the subsequent ones, and so on. Therefore, we can use the first 16 (0, 2)-sequence samples for the integrator samples for the first image sample for a pixel, then the next 16 for the next image sample, and so forth. The result is an extremely well-distributed set of sample points.

There are a handful of details that must be addressed before (0, 2)-sequences can be used in practice. The first is that we need to generate multiple sets of 2D sample values for each image sample, and we would like to generate different sample values in the areas around different pixels. One approach to this problem would be to use carefully chosen nonoverlapping subsequences of the (0, 2)-sequence for each pixel. Another approach, which is used in `pbrt`, is to randomly *scramble* the (0, 2)-sequence, giving a new (0, 2)-sequence built by randomly permuting the base b digits of the values in the original sequence.

The scrambling approach we will use is due to Kollig and Keller (2002). It repeatedly partitions and shuffles the unit square $[0, 1]^2$. In each of the two dimensions, it first divides the square in half, then swaps the two halves with 50% probability. Then, it splits each of the intervals [0, 0.5) and [0.5, 1) in half and randomly exchanges each of those two halves. This process continues recursively until floating-point precision intervenes and continuing the process would no longer change the computed values. This process was carefully designed so that it preserves the low-discrepancy properties of the set of points; otherwise, the advantages of the (0, 2)-sequence would be lost from the scrambling. Figure 7.28 shows an unscrambled (0, 2)-sequence and two randomly scrambled variations of it.

Two things make this process efficient: First, because we are scrambling two sequences that are computed in base 2, the digits a_i of the sequences are all 0 or 1, and scrambling a particular digit is equivalent to exclusive-ORing it with 0 or 1. Second, the simplification is made such that at each level l of the recursive scrambling, the same decision will be

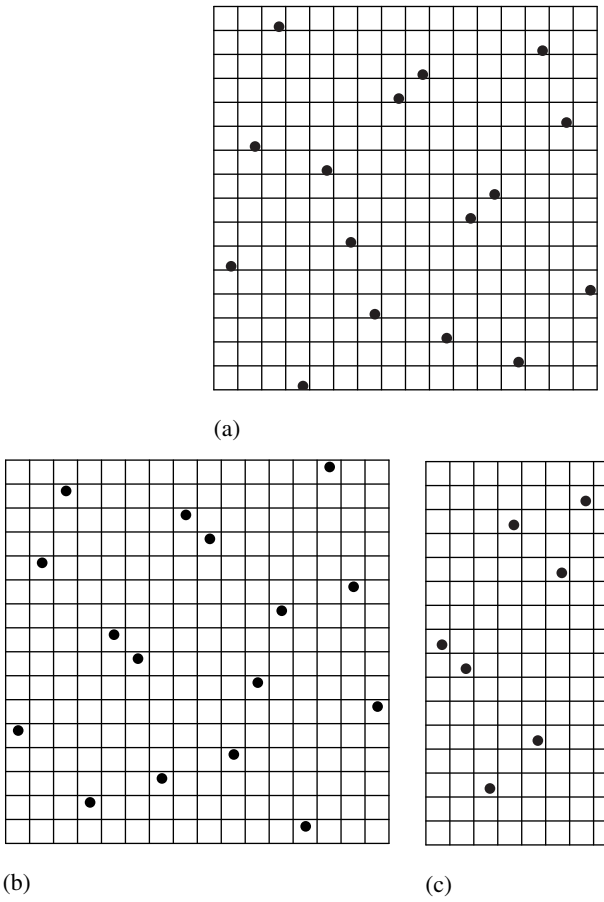


Figure 7.28: (a) A low-discrepancy (0, 2)-sequence-based sampling pattern and (b, c) two randomly scrambled instances of it. Random scrambling of low-discrepancy patterns is an effective way to eliminate the artifacts that would be present in images if we used the same sampling pattern in every pixel, while still preserving the low-discrepancy properties of the point set being used.

made as to whether to swap each of the 2^{l-1} pairs of subintervals or not. The result of these two design choices is that the scrambling can be encoded as a set of bits stored in a `uint32_t` and can be applied to the original digits via exclusive-OR operations.

The `Sample02()` function generates a sample from a scrambled (0, 2)-sequence using the given scramble values. The sequence used here is constructed from two 1D low-discrepancy sequences that together form a (0, 2)-sequence. An arbitrary pair of 1D low-discrepancy sequences will not necessarily form a (0, 2)-sequence, however.

⟨*Sampling Inline Functions*⟩ ≡
```
inline void Sample02(uint32_t n, const uint32_t scramble[2],
                     float sample[2]) {
    sample[0] = VanDerCorput(n, scramble[0]);
    sample[1] = Sobol2(n, scramble[1]);
}
```

The implementations of the van der Corput and Sobol′ low-discrepancy sequences here are also specialized for the base 2 case. Each of them takes a `uint32_t` value `scramble` that encodes a random permutation to apply and computes the `n`th value from each of the sequences as it simultaneously applies the permutation. It is worthwhile to convince yourself that the `VanDerCorput()` function computes the same values as `RadicalInverse()` in the preceding when they are called with a zero `scramble` and a `base` of two, respectively.

⟨*Sampling Inline Functions*⟩ +≡
```
inline float VanDerCorput(uint32_t n, uint32_t scramble) {
    ⟨Reverse bits of n 372⟩
    n ^= scramble;
    return ((n>>8) & 0xffffff) / float(1 << 24);
}
```

The bits of a 32-bit quantity can be efficiently reversed with a series of logical bit operations. The first line of the fragment below swaps the lower 16 bits with the upper 16 bits of the value. The next line simultaneously swaps the first 8 bits of the result with the second 8 bits and the third 8 bits with the fourth. This process continues until the last line, which swaps adjacent bits. To understand this code, it's helpful to write out the binary values of the various hexidecimal constants. For example, `0xff00ff00` is 11111111000000001111111100000000 in binary; it's easy to see that ORing with this value masks off the first and third 8-bit quantities.

⟨*Reverse bits of* n⟩ ≡ 372
```
n = (n << 16) | (n >> 16);
n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
```

⟨*Sampling Inline Functions*⟩ +≡
```
inline float Sobol2(uint32_t n, uint32_t scramble) {
    for (uint32_t v = 1 << 31; n != 0; n >>= 1, v ^= v >> 1)
        if (n & 0x1) scramble ^= v;
    return ((scramble>>8) & 0xffffff) / float(1 << 24);
}
```

7.4.4 THE LOW-DISCREPANCY SAMPLER

The `LDSampler` implements a low-discrepancy pattern based on (0, 2)-sequences, since they give the foundation for an effective approach for generating multiple samples per image sample for integration. Because a (0, 2)-sequence is being used, the number of samples for each pixel must be a power of two. It generates samples for positions on the

RadicalInverse() 362
Sobol2() 372
VanDerCorput() 372



(a) (b) (c)

Figure 7.29: Comparisons of the Stratified and Low-Discrepancy Samplers for Rendering Depth of Field. (a) A reference image of the blurred edge of an out-of-focus sphere, (b) an image rendered using the `StratifiedSampler`, and (c) an image using the `LDSampler`. The `LDSampler`'s results are better than the stratified image, although the difference is far less than the difference between stratified and random sampling.

lens and for the two-dimensional integrator samples with scrambled (0, 2)-sequences, and one-dimensional samples for time and integrators are generated with scrambled van der Corput sequences. Figure 7.29 compares the result of using an (0, 2)-sequence for sampling the lens for the depth of field to using a stratified pattern.

⟨*LDSampler Declarations*⟩ ≡
```
class LDSampler : public Sampler {
public:
    ⟨LDSampler Public Methods 374⟩
private:
    ⟨LDSampler Private Data 374⟩
};
```

The constructor rounds the number of samples per pixel up to a power of two if necessary, since subsets of (0, 2)-sequences that are not a power of two in size are much less well distributed over [0, 1] than those that are. Like the `StratifiedSampler`, the `LDSampler` uses a preallocated temporary storage buffer, `sampleBuf`, during sample generation. The amount of storage that must be allocated is computed in a separate utility function; thus, other samplers (such as the `AdaptiveSampler` below) can also use its functionality.

AdaptiveSampler 386
IsPowerOf2() 1001
LDSampler 373
LDSampler::nPixelSamples 374
LDSampler::sampleBuf 374
LDSampler::xPos 374
LDSampler::yPos 374
RoundUpPow2() 1002
Sampler 340
Sampler::xPixelStart 340
Sampler::yPixelStart 340
StratifiedSampler 349
Warning() 1005

⟨*LDSampler Method Definitions*⟩ ≡
```
LDSampler::LDSampler(int xstart, int xend, int ystart, int yend, int ps,
                     float sopen, float sclose)
    : Sampler(xstart, xend, ystart, yend, RoundUpPow2(ps), sopen, sclose) {
    xPos = xPixelStart;
    yPos = yPixelStart;
    if (!IsPowerOf2(ps)) {
        Warning("Pixel samples being rounded up to power of 2");
        nPixelSamples = RoundUpPow2(ps);
    } else
        nPixelSamples = ps;
    sampleBuf = NULL;
}
```

⟨*LDSampler Private Data*⟩ ≡ 373
```
int xPos, yPos, nPixelSamples;
float *sampleBuf;
```

The implementation of the `GetSubSampler()` method is quite similar to that of the `StratifiedSampler::GetSubSampler()` method, so it isn't included here.

⟨*LDSampler Public Methods*⟩ ≡ 373
```
Sampler *GetSubSampler(int num, int count);
```

As mentioned earlier, the low-discrepancy sequences used here need power-of-two sample sizes.

⟨*LDSampler Public Methods*⟩ +≡ 373
```
int RoundSize(int size) const { return RoundUpPow2(size); }
```

Usage of `xPos` and `yPos` is just like the corresponding member variables in the `StratifiedSampler`; pixels are scanned left-to-right and top-to-bottom in the sampling region, and an entire pixel's worth of samples are generated for each call to `GetMoreSamples()`. `LDPixelSample()` fills in the `samples` array with samples for the current pixel.

⟨*LDSampler Method Definitions*⟩ +≡
```
int LDSampler::GetMoreSamples(Sample *samples, RNG &rng) {
    if (yPos == yPixelEnd) return 0;
    if (sampleBuf == NULL)
        sampleBuf = new float[LDPixelSampleFloatsNeeded(samples,
                                                        nPixelSamples)];
    LDPixelSample(xPos, yPos, shutterOpen, shutterClose,
                  nPixelSamples, samples, sampleBuf, rng);
    if (++xPos == xPixelEnd) {
        xPos = xPixelStart;
        ++yPos;
    }
    return nPixelSamples;
}
```

`LDPixelSample()` uses a temporary buffer `buf` in its work. The memory that `buf` points to must be large enough to store all of the image, lens, time, and integrator samples for a pixel. This value is computed by `LDPixelSampleFloatsNeeded()`, given a `Sample` and the pixel sampling rate. Its implementation computes the total number of `floats` needed for a single pixel sample and then returns this value multiplied by the total number of samples per pixel.

⟨*Sampling Function Definitions*⟩ +≡
```
int LDPixelSampleFloatsNeeded(const Sample *sample, int nPixelSamples) {
    int n = 5; // 2 lens + 2 pixel + time
    for (uint32_t i = 0; i < sample->n1D.size(); ++i)
        n += sample->n1D[i];
    for (uint32_t i = 0; i < sample->n2D.size(); ++i)
        n += 2 * sample->n2D[i];
```

LDPixelSample() 375
LDPixelSampleFloatsNeeded() 374
LDSampler 373
LDSampler::nPixelSamples 374
LDSampler::sampleBuf 374
LDSampler::xPos 374
LDSampler::yPos 374
RNG 1003
RoundUpPow2() 1002
Sample 343
Sample::n1D 344
Sample::n2D 344
Sampler 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
StratifiedSampler 349
StratifiedSampler::
GetSubSampler() 350

```
    return nPixelSamples * n;
}
```

Given the large linear temporary buffer `buf`, `LDPixelSample()` first computes pointers into the buffer for the various types of samples. After generating the samples into this buffer, it finishes by copying them into the `samples` array before returning. These extra steps are needed because the sample generation utility routines are written to generate samples into flat arrays of `float`s, rather than directly into `Sample` structures.

```
⟨Sampling Function Definitions⟩ +≡
void LDPixelSample(int xPos, int yPos, float shutterOpen,
        float shutterClose, int nPixelSamples, Sample *samples,
        float *buf, RNG &rng) {
    ⟨Prepare temporary array pointers for low-discrepancy camera samples 375⟩
    ⟨Prepare temporary array pointers for low-discrepancy integrator samples 375⟩
    ⟨Generate low-discrepancy pixel samples 376⟩
    ⟨Initialize samples with computed sample values 376⟩
}
```

For convenience, separate pointers are initialized from the temporary buffer to store the initial image, lens, and time samples.

```
⟨Prepare temporary array pointers for low-discrepancy camera samples⟩ ≡ 375
float *imageSamples = buf; buf += 2 * nPixelSamples;
float *lensSamples = buf;  buf += 2 * nPixelSamples;
float *timeSamples = buf;  buf += nPixelSamples;
```

Next, the `oneDSamples` and `twoDSamples` arrays are set up so that their entries point to memory from the temporary buffer to store their respective sample values.

```
⟨Prepare temporary array pointers for low-discrepancy integrator samples⟩ ≡ 375
uint32_t count1D = samples[0].n1D.size();
uint32_t count2D = samples[0].n2D.size();
const uint32_t *n1D = count1D > 0 ? &samples[0].n1D[0] : NULL;
const uint32_t *n2D = count2D > 0 ? &samples[0].n2D[0] : NULL;
float **oneDSamples = ALLOCA(float *, count1D);
float **twoDSamples = ALLOCA(float *, count2D);
for (uint32_t i = 0; i < count1D; ++i) {
    oneDSamples[i] = buf;
    buf += n1D[i] * nPixelSamples;
}
for (uint32_t i = 0; i < count2D; ++i) {
    twoDSamples[i] = buf;
    buf += 2 * n2D[i] * nPixelSamples;
}
```

ALLOCA() 1009
RNG 1003
Sample 343
Sample::n1D 344
Sample::n2D 344

Sample values can now be computed. The `LDShuffleScrambled*D()` utility routines, to be defined shortly, compute one- and two-dimensional scrambled low-discrepancy sampling patterns.

⟨*Generate low-discrepancy pixel samples*⟩ ≡ 375
```
LDShuffleScrambled2D(1, nPixelSamples, imageSamples, rng);
LDShuffleScrambled2D(1, nPixelSamples, lensSamples, rng);
LDShuffleScrambled1D(1, nPixelSamples, timeSamples, rng);
for (uint32_t i = 0; i < count1D; ++i)
    LDShuffleScrambled1D(n1D[i], nPixelSamples, oneDSamples[i], rng);
for (uint32_t i = 0; i < count2D; ++i)
    LDShuffleScrambled2D(n2D[i], nPixelSamples, twoDSamples[i], rng);
```

Finally, once the sample values have been computed, the `samples` array can be filled in. For image and time samples, the values have to be be mapped to the appropriate range; other sample types can be copied directly.

⟨*Initialize* `samples` *with computed sample values*⟩ ≡ 375
```
for (int i = 0; i < nPixelSamples; ++i) {
    samples[i].imageX = xPos + imageSamples[2*i];
    samples[i].imageY = yPos + imageSamples[2*i+1];
    samples[i].time = Lerp(timeSamples[i], shutterOpen, shutterClose);
    samples[i].lensU = lensSamples[2*i];
    samples[i].lensV = lensSamples[2*i+1];
    ⟨Copy integrator samples into samples[i] 376⟩
}
```

The array indexing to copy samples for the integrators is a little more tricky. The `j`th element of the `oneDSamples` array, for example, points to the start of an array of `nPixelSamples * n1D[j]` floating-point values, where the first `n1D[j]` of them are for the first pixel sample, the next `n1D[j]` are for the next pixel sample, and so forth. The code below handles copying the appropriate values into the individual samples.

⟨*Copy integrator samples into* `samples[i]`⟩ ≡ 376
```
for (uint32_t j = 0; j < count1D; ++j) {
    int startSamp = n1D[j] * i;
    for (uint32_t k = 0; k < n1D[j]; ++k)
        samples[i].oneD[j][k] = oneDSamples[j][startSamp+k];
}
for (uint32_t j = 0; j < count2D; ++j) {
    int startSamp = 2 * n2D[j] * i;
    for (uint32_t k = 0; k < 2*n2D[j]; ++k)
        samples[i].twoD[j][k] = twoDSamples[j][startSamp+k];
}
```

There is a subtle implementation detail that must be accounted for in using scrambled (0, 2)-sequences in practice.[7] Often, integrators will use samples from more than one of the sampling patterns that the sampler creates in the process of computing the values of particular integrals. For example, they might use a sample from a one-dimensional pattern to select one of the N light sources in the scene to sample illumination from, and

CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
LDSampler::nPixelSamples 374
LDSampler::xPos 374
LDSampler::yPos 374
LDShuffleScrambled1D() 377
LDShuffleScrambled2D() 377
Lerp() 1000
Sampler::shutterClose 340
Sampler::shutterOpen 340

7 Indeed, the importance of this issue wasn't fully appreciated by the authors until after going through the process of debugging some unexpected noise patterns in rendered images when this sampler was being used.

then might use a sample from a two-dimensional pattern to select a sample point on that light source, if it is an area light.

Even if these two patterns are computed with random scrambling with different random scramble values for each one, some correlation can still remain between elements of these patterns, such that the *i*th element of the one-dimensional pattern and the *i*th element of the two-dimensional pattern are related. As such, in the earlier area lighting example, the distribution of sample points on each light source would not in general cover the entire light due to this correlation, leading to unusual rendering errors.

This problem can be solved easily enough by randomly shuffling the various patterns individually after they are generated. For example, the `LDShuffleScrambled1D()` function first generates a scrambled one-dimensional low-discrepancy sampling pattern, giving a well-distributed set of samples across all of the image samples for this pixel. Then, it shuffles these samples in two ways. Consider, for example, a pixel with 8 image samples, each of which has 4 1D samples for the integrator (giving a total of 32 integrator samples). First, it shuffles samples within each of the 8 groups of 4 samples, putting each set of 4 into a random order. Next, it shuffles each of the 8 groups of 4 samples as a block, with respect to the other blocks of 4 samples.

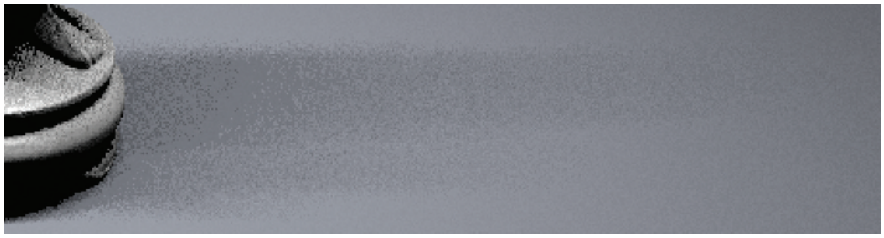
⟨*Sampling Inline Functions*⟩ +≡
```
inline void LDShuffleScrambled1D(int nSamples, int nPixel,
                                 float *samples, RNG &rng) {
    uint32_t scramble = rng.RandomUInt();
    for (int i = 0; i < nSamples * nPixel; ++i)
        samples[i] = VanDerCorput(i, scramble);
    for (int i = 0; i < nPixel; ++i)
        Shuffle(samples + i * nSamples, nSamples, 1, rng);
    Shuffle(samples, nPixel, nSamples, rng);
}
```

⟨*Sampling Inline Functions*⟩ +≡
```
inline void LDShuffleScrambled2D(int nSamples, int nPixel,
                                 float *samples, RNG &rng) {
    uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
    for (int i = 0; i < nSamples * nPixel; ++i)
        Sample02(i, scramble, &samples[2*i]);
    for (int i = 0; i < nPixel; ++i)
        Shuffle(samples + 2 * i * nSamples, nSamples, 2, rng);
    Shuffle(samples, nPixel, 2 * nSamples, rng);
}
```

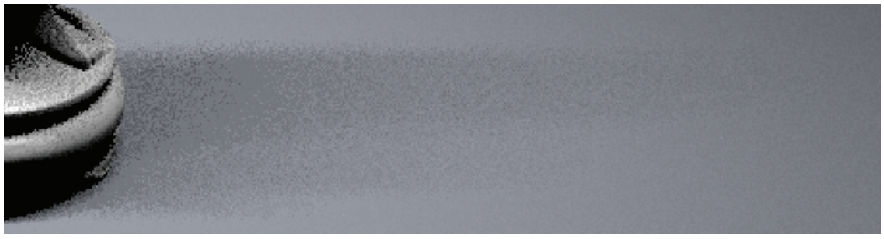
LDSampler::nPixelSamples 374
LDShuffleScrambled1D() 377
RNG 1003
RNG::RandomUInt() 1003
Sample02() 372
Shuffle() 354
VanDerCorput() 372

Given the implementation of `GetMoreSamples()` above, the maximum number of samples returned is just the total number of samples for each pixel.

⟨*LDSampler Public Methods*⟩ +≡ 373
```
int MaximumSampleCount() { return nPixelSamples; }
```



(a)



(b)

Figure 7.30: When the `LDSampler` is used for the area light sampling example, similar results are generated (a) with both 1 image sample and 16 light samples as well as (b) with 16 image samples and 1 light sample, thanks to the (0, 2)-sequence sampling pattern that ensures good distribution of samples over the pixel area in both cases. Compare these images to Figure 7.23, where the stratified pattern generates a much worse set of light samples when only 1 light sample is taken for each of the 16 image samples.

Figure 7.30 shows the result of using the (0, 2)-sequence for the area lighting example scene. Note that not only does it give a visibly better image than stratified patterns, but it also does well with one light sample per image sample, unlike the stratified sampler.

⋆ 7.5 BEST-CANDIDATE SAMPLING PATTERNS

A shortcoming of both the `StratifiedSampler` and the `LDSampler` is that they both generate good image sample patterns around a single pixel, but neither has any mechanism for ensuring that the image samples at adjacent pixels are well distributed with respect to the samples at the current pixel. For example, we would like to avoid having two adjacent pixels choose samples that are very close to their shared edge. The *Poisson disk pattern* addresses this issue of well-separated sample placement and has been shown to be an excellent image sampling pattern. The Poisson disk pattern is a group of points with no two of them closer to each other than some specified distance. Studies have shown that the rods and cones in the eye are distributed in a similar way, which suggests that this pattern might be effective for imaging. An easy way to generate Poisson disk patterns is with *dart throwing*: a program generates samples randomly, throwing away any that are closer to a previous sample than a fixed threshold distance. This can be a very expensive process, since many darts may need to be thrown before one is accepted. The "Further

`LDSampler` 373
`StratifiedSampler` 349

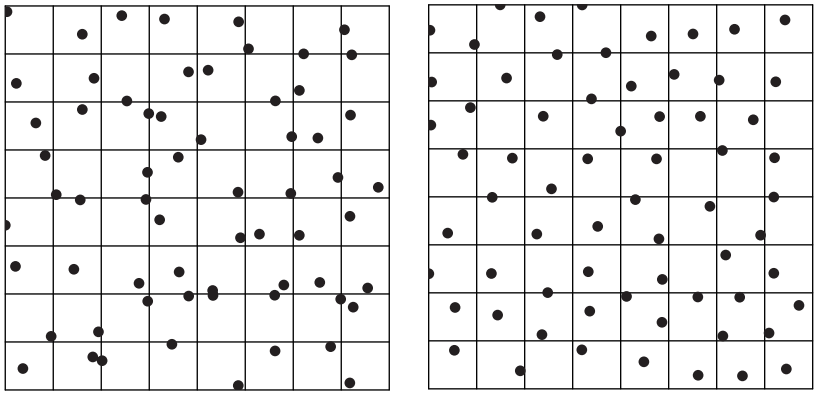


Figure 7.31: Comparison of Sampling Patterns. (a) A jittered pattern: note clumping of samples and undersampling in some areas. (b) A pattern generated with the best-candidate algorithm: it is effectively as good as the Poisson disk pattern.

Reading" section at the end of the chapter has pointers to recent work on more efficient approaches.

A related approach due to Don Mitchell is the *best-candidate* algorithm. Each time a new sample is to be computed, a large number of random candidates are generated. All of these candidates are compared to the previous samples, and the one that is farthest away from all of the previous ones is added to the pattern. Although this algorithm doesn't guarantee the Poisson disk property, it usually does quite well at finding well-separated points if enough candidates are generated. It has the additional advantage that any prefix of the final pattern is itself a well-distributed sampling pattern. Furthermore, the best-candidate algorithm makes it easier to generate a good pattern with a predetermined number of samples than is possible with the dart-throwing algorithm. A comparison of a stratified pattern to a best-candidate pattern is shown in Figure 7.31.

In this section we will present a `Sampler` implementation that uses the best-candidate algorithm to compute sampling patterns that also have good distributions of samples in the additional sampling dimensions. Because generating the sample positions is computationally intensive, we have computed a good sampling pattern once in a preprocess. The pattern encodes 5D image, time, and lens sample positions; it is stored in a table that can be efficiently used at rendering time. Rather than computing a sampling pattern large enough to sample the largest image we'd ever render, the pattern can be reused by tiling it over the image plane. This means that the pattern must have *toroidal topology*—when computing the distance between two samples, we must compute the distance between them as if the square sampling region was rolled into a torus. The code that computed the pattern can be found in the file `tools/samplepat.cpp`; the results are stored in the file `samplers/bestcandidate.out`.

The BestCandidateSampler is the Sampler that uses the best-candidate sample table. If an average of nPixelSamples samples is to be taken in each pixel, then a single copy of the sample table covers SQRT_SAMPLE_TABLE_SIZE / sqrt(nPixelSamples) pixels in the x and y directions.

⟨*BestCandidate Sampling Constants*⟩ ≡
```
#define SQRT_SAMPLE_TABLE_SIZE 64
#define SAMPLE_TABLE_SIZE (SQRT_SAMPLE_TABLE_SIZE * \
                           SQRT_SAMPLE_TABLE_SIZE)
```

We can envision the entire 2D image plane as being covered by replicated instances of this sample table; we will arbitrarily say that the upper left corner of one instance of it lies at the (0, 0) pixel location and that the placement of the other instances follows from there. Thus, if the width (and height) of the table is w, then pixels from (0, 0) to (w, w) are covered by the tile with indices (0, 0), pixels from $(w, 0)$ to $(2w, w)$ are covered by the tile (1, 0), and so forth. Specifically, the mapping from a continuous pixel location (x, y) to the 2D index (i, j) of the sample table that it lies within is given by

$$(i, j) = (\lfloor x/\text{tableWidth} \rfloor, \lfloor y/\text{tableWidth} \rfloor).$$

Figure 7.32 illustrates this idea.

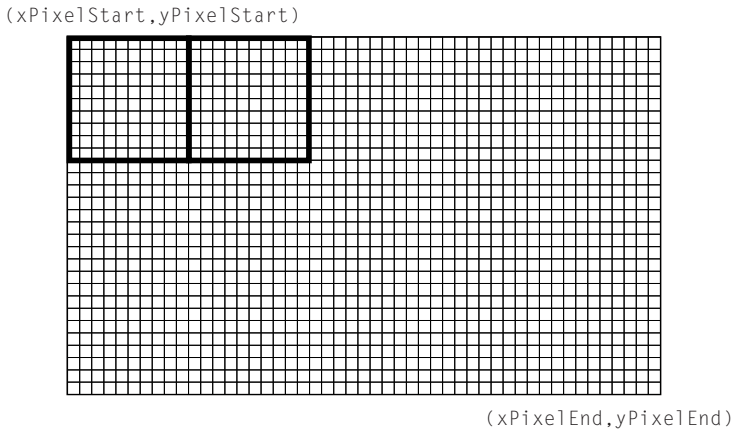


Figure 7.32: Because it is comprised of thousands of samples, the best-candidate sample table will generally cover a large number of pixels. Each instance of the table covers a width and height of some number of pixels, n, based on the size of the table and the number of samples wanted per pixel. One of the main tasks of the BestCandidateSampler is to tile this pattern over the image plane, mapping its sample positions to image sample positions.

BestCandidateSampler 381
Sampler 340
SQRT_SAMPLE_TABLE_SIZE 380

⟨*BestCandidateSampler Declarations*⟩ ≡
```
class BestCandidateSampler : public Sampler {
public:
    ⟨BestCandidateSampler Public Methods 381⟩
private:
    ⟨BestCandidateSampler Private Data 381⟩
};
```

The `tableWidth` member variable stores the raster space width in pixels that the precomputed sample table spans. The constructor then computes the range of sample table indices that must be considered to find all of the samples in the sampler's pixel region. (Because the samples within the sample table don't have any spatial organization, we need to check each one of them to see if it lies within the image tile that samples are being generated for if the tile extent at all overlaps the sampler's pixel region.) This range is stored in the member variables `xTileStart`, `xTileEnd`, `yTileStart`, and `yTileEnd`.

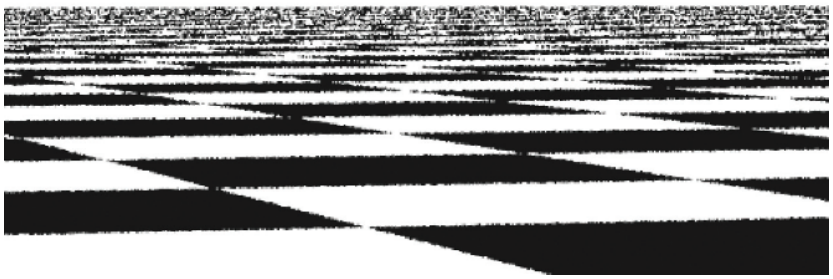
`xTile` and `yTile` hold the current integer indices of the sample table instance being used, and `tableOffset` holds the current offset into the sample table; when it is advanced to the point where it reaches the end of the table, the sampler advances to the next region of the image that the table covers by scanning `xTile` left to right and then `yTile` top to bottom.

Figure 7.33 compares the result of using the best-candidate pattern to the stratified pattern for rendering the checkerboard. Figure 7.34 shows the result of using this pattern for depth of field. For the number of samples used in that figure, the low-discrepancy sampler gives a better result, likely because the best candidate precomputation step searches around a fixed-size region of pixel samples when selecting lens samples. Depending on the actual number of pixel samples used, this region may map to much less or much more than a single pixel area.

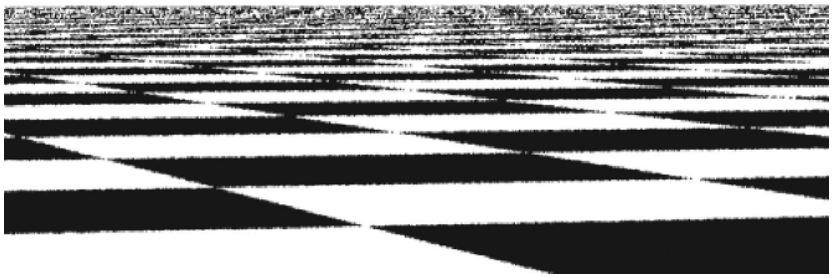
⟨*BestCandidateSampler Public Methods*⟩ ≡ 381
```
BestCandidateSampler(int xstart, int xend, int ystart, int yend,
                     int nPixelSamples, float sopen, float sclose)
    : Sampler(xstart, xend, ystart, yend, nPixelSamples, sopen, sclose) {
    tableWidth = (float)SQRT_SAMPLE_TABLE_SIZE /
                 (float)sqrtf(nPixelSamples);
    xTileStart = Floor2Int(xstart / tableWidth);
    xTileEnd = Floor2Int(xend / tableWidth);
    yTileStart = Floor2Int(ystart / tableWidth);
    yTileEnd = Floor2Int(yend / tableWidth);
    xTile = xTileStart;
    yTile = yTileStart;
    tableOffset = 0;
}
```

⟨*BestCandidateSampler Private Data*⟩ ≡ 381
```
float tableWidth;
int tableOffset;
int xTileStart, xTileEnd, yTileStart, yTileEnd;
int xTile, yTile;
```

BestCandidateSampler 381
BestCandidateSampler::
tableOffset 381
BestCandidateSampler::
tableWidth 381
BestCandidateSampler::
xTile 381
BestCandidateSampler::
xTileEnd 381
BestCandidateSampler::
xTileStart 381
BestCandidateSampler::
yTile 381
BestCandidateSampler::
yTileEnd 381
BestCandidateSampler::
yTileStart 381
Sampler 340
SQRT_SAMPLE_TABLE_SIZE 380



(a)



(b)

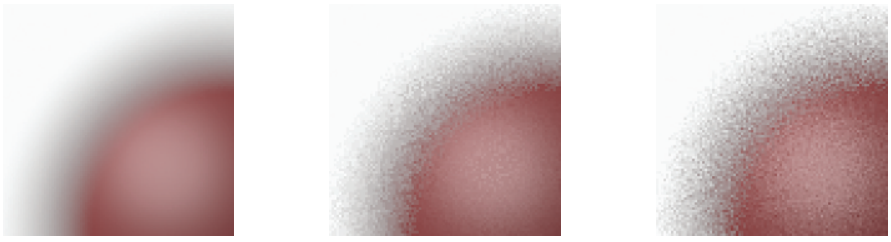


(c)



(d)

Figure 7.33: Comparison of the Stratified Sampling Pattern with the Best-Candidate Sampling Pattern. (a) The stratified pattern with a single sample per pixel. (b) The best-candidate pattern with a single sample per pixel. (c) The stratified pattern with four samples per pixel. (d) The four-sample best-candidate pattern. Although the differences are subtle, note that the edges of the checks in the foreground are less aliased when the best-candidate pattern is used, and it also does better at resolving the checks toward the horizon, particularly on the sides of the image. Furthermore, the noise from the best-candidate pattern tends to be higher frequency, and therefore more visually acceptable.



(a) (b) (c)

Figure 7.34: (a) Reference depth of field image, and images rendered with (b) the low-discrepancy and (c) best-candidate samplers. Here the low-discrepancy sampler is again the most effective.

Here we incorporate the sample data stored in `samplers/bestcandidate.out`.

⟨*BestCandidateSampler Private Data*⟩ +≡ 381
```
static const float sampleTable[SAMPLE_TABLE_SIZE][5];
```

⟨*BestCandidateSampler Method Definitions*⟩ ≡
```
#include "samplers/bestcandidate.out"
```

The implementations of the `GetSubSampler()`, `RoundSize()`, and `MaximumSampleCount()` methods are straightforward and similar to those of other previously defined samplers in this chapter.

The `BestCandidateSampler::GetMoreSamples()` method returns a single sample for each best-candidate sample point that is inside the sampler's extent.

⟨*BestCandidateSampler Method Definitions*⟩ +≡
```
int BestCandidateSampler::GetMoreSamples(Sample *sample, RNG &rng) {
again:
    if (tableOffset == SAMPLE_TABLE_SIZE) {
        ⟨Advance to next best-candidate sample table position 384⟩
    }
    ⟨Compute raster sample from table 385⟩
    ⟨Check sample against crop window, goto again if outside 385⟩
    ⟨Compute integrator samples for best-candidate sample 385⟩
    ++tableOffset;
    return 1;
}
```

BestCandidateSampler 381
BestCandidateSampler::GetMoreSamples() 383
BestCandidateSampler::tableOffset 381
RNG 1003
Sample 343
SAMPLE_TABLE_SIZE 380

If it has reached the end of the sample table, the sampler tries to move forward to the next precomputed sample tile, scanning across tiles in x and then in y.

⟨*Advance to next best-candidate sample table position*⟩ ≡ 383
```
tableOffset = 0;
if (++xTile > xTileEnd) {
    xTile = xTileStart;
    if (++yTile > yTileEnd)
        return 0;
}
⟨Update sample shifts 384⟩
```

One problem with using tiled sample patterns is that there may be subtle image artifacts aligned with the edges of the pattern on the image plane due to the same values being used repeatedly for time and lens position in each replicated sample region. Not only are the same samples used and reused (whereas the `StratifiedSampler` and `LDSampler` will at least generate different time and lens values for each image sample), but the upper-left sample in each block of samples will always have the same time and lens values, and so on.

A solution to this problem is to transform the set of sample values each time we reuse the pattern. This can be done using *Cranley–Patterson rotations*, which compute

$$X_i' = (X_i + \xi_i) \bmod 1$$

in each dimension, where X_i is the sample value and ξ_i is a random number between zero and one. Because the time and lens portions of sampling patterns were computed with toroidal topology, where the distance between two candidate samples was computed such that, for example, the distance between the points .01 and .99 was computed to be .02, rather than .98, the resulting pattern is still well distributed and seamless. The table of random offsets for time and lens position ξ_i is updated each time the sample pattern is reused.

An important detail is that the `RNG` used to compute the random offsets is seeded with a value derived from the current tile indices; this ensures that the same offsets are used when a best-candidate sample tile spans multiple image tiles used for parallelization.

⟨*Update sample shifts*⟩ ≡ 384
```
RNG tileRng(xTile + (yTile<<8));
for (int i = 0; i < 3; ++i)
    sampleOffsets[i] = tileRng.RandomFloat();
```

⟨*BestCandidateSampler Private Data*⟩ +≡ 381
```
float sampleOffsets[3];
```

Computing the raster space sample position from the positions in the table requires some simple indexing and scaling. We don't use the Cranley–Patterson shifting technique on image samples because this would cause the sampling points at the borders between repeated instances of the table to have a poor distribution. Preserving good image distribution is more important than reducing correlation. The rest of the camera dimensions do use the shifting technique; the `WRAP` macro ensures that the result stays between zero and one.

BestCandidateSampler::
sampleOffsets 384
BestCandidateSampler::
xTile 381
BestCandidateSampler::
yTile 381
LDSampler 373
RNG 1003
RNG::RandomFloat() 1003
StratifiedSampler 349

⟨*Compute raster sample from table*⟩ ≡ 383
```
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
sample->imageX = (xTile + sampleTable[tableOffset][0]) * tableWidth;
sample->imageY = (yTile + sampleTable[tableOffset][1]) * tableWidth;
sample->time  = Lerp(WRAP(sampleOffsets[0] + sampleTable[tableOffset][2]),
                     shutterOpen, shutterClose);
sample->lensU = WRAP(sampleOffsets[1] +
                     sampleTable[tableOffset][3]);
sample->lensV = WRAP(sampleOffsets[2] +
                     sampleTable[tableOffset][4]);
```

Some of the generated samples will generally be outside the sampler's pixel sample region. The sampler detects this case by checking the sample against the region of pixels to be sampled and generating a new sample if it's out of bounds.

⟨*Check sample against crop window, goto* `again` *if outside*⟩ ≡ 383
```
if (sample->imageX < xPixelStart || sample->imageX >= xPixelEnd ||
    sample->imageY < yPixelStart || sample->imageY >= yPixelEnd) {
    ++tableOffset;
    goto again;
}
```

If the sample is inside the sampler's pixel region, samples for the integrators are generated using shuffled (0, 2)-sequences.

⟨*Compute integrator samples for best-candidate sample*⟩ ≡ 383
```
for (uint32_t i = 0; i < sample->n1D.size(); ++i)
     LDShuffleScrambled1D(sample->n1D[i], 1, sample->oneD[i], rng);
for (uint32_t i = 0; i < sample->n2D.size(); ++i)
     LDShuffleScrambled2D(sample->n2D[i], 1, sample->twoD[i], rng);
```

7.6 ADAPTIVE SAMPLING

The `AdaptiveSampler` implements a simple adaptive sampling algorithm that attempts to more-efficiently generate high-quality images by adding extra samples in parts of the image that are more complex than others. Recall the basic structure of the interaction between `Sampler`s and the main rendering loop in `SamplerRendererTask::Run()` in Section 1.3.4

1. The rendering loop calls `Sampler::GetMoreSamples()` to get a collection of samples.
2. The `SamplerRendererTask` computes the radiance values along the corresponding rays.
3. The samples, radiance values, and intersection information are passed back to the `Sampler::ReportResults()` method.
4. Based on the Boolean return value from `ReportResults()`, the `SamplerRendererTask` either discards the set of samples in preparation for a new set or passes them along to the `Film` to be incorporated into the final image.

AdaptiveSampler 386
BestCandidateSampler::sampleOffsets 384
BestCandidateSampler::sampleTable 383
BestCandidateSampler::tableWidth 381
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
Film 403
LDShuffleScrambled1D() 377
LDShuffleScrambled2D() 377
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
Sampler 340
Sampler::GetMoreSamples() 340
Sampler::ReportResults() 341
Sampler::shutterClose 340
Sampler::shutterOpen 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340
SamplerRendererTask::Run() 30

This interaction allows the sampler to examine the results for a collection of samples and decide to take more samples when appropriate. The values passed to `ReportResults()` can be used in a variety of ways to make this decision.

One challenge with adaptive sampling is finding good criteria for deciding when to take more samples. The challenge is that the information the sampler has to work with is limited: as usual, all it has available to it is the values of a set of point samples, with no information about the behavior of the image function at other locations. The `AdaptiveSampler`, defined in the files `samplers/adaptive.h` and `samplers/adaptive.cpp`, supports two simple refinement criteria. The first is based on checking to see if different shapes are intersected by different samples, which indicates a likely geometric discontinuity, and the second checks for excessive contrast between the colors of different samples.

⟨*AdaptiveSampler Declarations*⟩ ≡
```
class AdaptiveSampler : public Sampler {
public:
    ⟨AdaptiveSampler Public Methods⟩
private:
    ⟨AdaptiveSampler Private Methods⟩
    ⟨AdaptiveSampler Private Data 386⟩
};
```

The `AdaptiveSampler` constructor isn't included here; its main task is to initialize the following member variables. Like most of the other samplers in this chapter, it tracks the current pixel being sampled in `(xPos, yPos)`. It generates at least `minSamples`, but possibly as many as `maxSamples` in each pixel's extent. The `sampleBuf` buffer is used for temporary storage during sample generation, and the `method` enumerant denotes which refinement criteria it will use. Finally, `supersamplePixel` indicates whether the current pixel needs extra samples; it is initialized to `false`.

⟨*AdaptiveSampler Private Data*⟩ ≡ 386
```
int xPos, yPos;
int minSamples, maxSamples;
float *sampleBuf;
enum AdaptiveTest { ADAPTIVE_COMPARE_SHAPE_ID,
                    ADAPTIVE_CONTRAST_THRESHOLD };
AdaptiveTest method;
bool supersamplePixel;
```

The `GetSubSampler()`, `RoundSize()`, and `MaxmimumSampleCount()` methods are essentially the same as those for the `LDSampler`.

In addition to recording the current pixel being sampled, the `AdaptiveSampler` also carries state that reflects whether the pixel needs additional samples in `AdaptiveSampler::supersamplePixel`. The `GetMoreSamples()` and `ReportResults()` methods interact to set it as follows:

- Initially, `supersamplePixel` is `false` and the intial set of `minSamples` is generated for the current pixel by `GetMoreSamples()`. The rendering loop evaluates those sam-

AdaptiveSampler 386
AdaptiveSampler::
supersamplePixel 386
LDSampler 373
Sampler 340

ples and reports them back to ReportResults(). If further sampling is indicated, ReportResults() sets supersamplePixel to true and leaves (xPos, yPos) unchanged. Thus, the next call to GetMoreSamples() will generate a new set of maxSamples samples for the pixel. When these samples are provided to ReportResults(), supersamplePixel is reset to false and the current pixel is advanced. The implementation here only does a single stage of adaptive sampling, rather than sampling more if the results continue to vary excessively.

- If further sampling after the initial set of samples isn't needed, then ReportResults() leaves supersamplePixel as false and advances the current pixel coordinates as the other samplers do.

⟨*AdaptiveSampler Method Definitions*⟩ ≡

```
int AdaptiveSampler::GetMoreSamples(Sample *samples, RNG &rng) {
    if (!sampleBuf)
        sampleBuf = new float[LDPixelSampleFloatsNeeded(samples,
                                                        maxSamples)];
    if (supersamplePixel) {
        LDPixelSample(xPos, yPos, shutterOpen, shutterClose, maxSamples,
                      samples, sampleBuf, rng);
        return maxSamples;
    }
    else {
        if (yPos == yPixelEnd) return 0;
        LDPixelSample(xPos, yPos, shutterOpen, shutterClose, minSamples,
                      samples, sampleBuf, rng);
        return minSamples;
    }
}
```

AdaptiveSampler 386
AdaptiveSampler::
maxSamples 386
AdaptiveSampler::
minSamples 386
AdaptiveSampler::
needsSupersampling() 388
AdaptiveSampler::
sampleBuf 386
AdaptiveSampler::
supersamplePixel 386
AdaptiveSampler::xPos 386
AdaptiveSampler::yPos 386
Intersection 186
LDPixelSample() 375
LDPixelSampleFloatsNeeded() 374
RayDifferential 69
RNG 1003
Sample 343
Sampler::shutterClose 340
Sampler::shutterOpen 340
Sampler::yPixelEnd 340
Spectrum 263

Other than maintaining the appropriate state in (xPos, yPos) and supersamplePixel, the bulk of the work for ReportResults() is done by the needsSupersampling() utility method.

⟨*AdaptiveSampler Method Definitions*⟩ +≡

```
bool AdaptiveSampler::ReportResults(Sample *samples,
        const RayDifferential *rays, const Spectrum *Ls,
        const Intersection *isects, int count) {
    if (supersamplePixel) {
        supersamplePixel = false;
        ⟨Advance to next pixel for sampling for AdaptiveSampler 388⟩
        return true;
    }
    else if (needsSupersampling(samples, rays, Ls, isects, count)) {
        supersamplePixel = true;
        return false;
    }
```

```
        else {
            ⟨Advance to next pixel for sampling for AdaptiveSampler 388⟩
            return true;
        }
    }
```

⟨*Advance to next pixel for sampling for* AdaptiveSampler⟩ ≡ 387

```
    if (++xPos == xPixelEnd) {
        xPos = xPixelStart;
        ++yPos;
    }
```

As described above, the AdaptiveSampler supports two different criteria for deciding when to take more samples; the method enumerant indicates which one is used.

⟨*AdaptiveSampler Method Definitions*⟩ +≡

```
    bool AdaptiveSampler::needsSupersampling(Sample *samples,
            const RayDifferential *rays, const Spectrum *Ls,
            const Intersection *isects, int count) {
        switch (method) {
        case ADAPTIVE_COMPARE_SHAPE_ID:
            ⟨See if any shape ids differ within samples 388⟩
        case ADAPTIVE_CONTRAST_THRESHOLD:
            ⟨Compare contrast of sample differences to threshold 389⟩
        }
        return false;
    }
```

A simple test is to see if any of the shape ids for the intersected objects are different. (Recall from Section 3.1 that each Shape is assigned a unique 32-bit identifier.) If some of the rays intersect different Shapes, then it is likely that there are edges in the pixel's extent, such that further sampling may be useful.

This test is somewhat effective but fails in a number of important cases. For example, it's possible to have different shape ids but not have a geometric edge that needs additional sampling (consider two coplanar triangles with the same material that exactly share an edge). Alternatively, a complex Shape like a parametric patch can fold over itself as seen from the camera and have edges with respect to itself that could use more sampling. This criterion also doesn't capture the image function discontinuities due to hard shadow boundaries, which can be just as objectionable as geometric aliasing.

⟨*See if any shape ids differ within samples*⟩ ≡ 388

```
    for (int i = 0; i < count-1; ++i)
        if (isects[i].shapeId != isects[i+1].shapeId ||
            isects[i].primitiveId != isects[i+1].primitiveId)
            return true;
    return false;
```

A different criterion that addresses some of the above shortcomings is to compute the contrast between the colors of the radiance carried by the samples. If the contrast is exces-

AdaptiveSampler::method 386
AdaptiveSampler::xPos 386
AdaptiveSampler::yPos 386
Intersection 186
Intersection::primitiveId 186
Intersection::shapeId 186
RayDifferential 69
Sample 343
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Shape 108
Spectrum 263

sive, we assume that the pixel would benefit from additional sampling. Here, the average luminance of all of the samples is stored in `Lavg` and then the contrast is computed by finding the relative difference between each sample and the average. If this value is above a threshold, then additional samples are taken.

This test is not always successful, either. One particular problem comes from the interaction with `Textures` that filter high frequencies in their values to avoid aliasing. (The `ImageTexture` is a good example, where image maps are filtered based on the rate at which they are being sampled in order to eliminate high frequencies.) With a complex texture function, the contrast metric may indicate the need for more sampling even though the underlying image function actually doesn't suffer from aliasing.

⟨*Compare contrast of sample differences to threshold*⟩ ≡ 388

```
float Lavg = 0.f;
for (int i = 0; i < count; ++i)
    Lavg += Ls[i].y();
Lavg /= count;
const float maxContrast = 0.5f;
for (int i = 0; i < count; ++i)
    if (fabsf(Ls[i].y() - Lavg) / Lavg > maxContrast)
        return true;
return false;
```

Figure 7.35 shows the operation of these two adaptive sampling criteria with the contemporary house interior scene; each image has a white pixel for pixels that were supersampled and a black pixel otherwise.

7.7 IMAGE RECONSTRUCTION

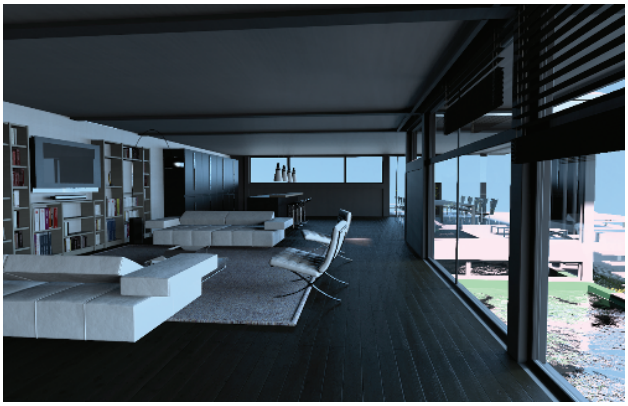
Given carefully chosen image samples, we need to convert the samples and their computed radiance values into pixel values for display or storage. According to signal processing theory, we need to do three things to compute final values for each of the pixels in the output image:

1. Reconstruct a continuous image function $\tilde{L}$ from the set of image samples.
2. Prefilter the function $\tilde{L}$ to remove any frequencies past the Nyquist limit for the pixel spacing.
3. Sample $\tilde{L}$ at the pixel locations to compute the final pixel values.

Because we know that we will be resampling the function $\tilde{L}$ at only the pixel locations, it's not necessary to construct an explicit representation of the function. Instead, we can combine the first two steps using a single filter function.

Recall that if the original function had been uniformly sampled at a frequency greater than the Nyquist frequency and reconstructed with the sinc filter, then the reconstructed function in the first step would match the original image function perfectly—quite a feat since we only have point samples. But because the image function almost always will have higher frequencies than could be accounted for by the sampling rate (due to edges, etc.), we chose to sample it nonuniformly, trading off noise for aliasing.

`ImageTexture` 524
`Spectrum::y()` 273
`Texture` 519



(a)



(b)



(c)

Figure 7.35: Adaptive Sampling Criteria in Action. (a) Example scene, rendered normally. (b) Pixels selected for adaptive sampling when more than one shape is intersected by the rays from the original set of samples. (c) Pixels selected for adaptive sampling when using contrast between the samples' radiance values to decide when to take more samples. *(Model courtesy of Florent Boyer.)*

The theory behind ideal reconstruction depends on the samples being uniformly spaced. While a number of attempts have been made to extend the theory to nonuniform sampling, there is not yet an accepted approach to this problem. Furthermore, because the sampling rate is known to be insufficient to capture the function, perfect reconstruction isn't possible. Recent research in the field of sampling theory has revisited the issue of reconstruction with the explicit acknowledgment that perfect reconstruction is not generally attainable in practice. With this slight shift in perspective has come powerful new techniques for reconstruction. See, for example, Unser (2000) for a survey of these developments. In particular, the goal of research in reconstruction theory has shifted from perfect reconstruction to developing reconstruction techniques that can be shown to minimize error between the reconstructed function and the original function, *regardless of whether the original was band limited*.

While the reconstruction techniques used in `pbrt` are not directly built on these new approaches, they serve to explain the experience of practitioners that applying perfect reconstruction techniques to samples taken for image synthesis generally does not result in the highest-quality images.

To reconstruct pixel values, we will consider the problem of interpolating the samples near a particular pixel. To compute a final value for a pixel $I(x, y)$, interpolation results in computing a weighted average

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}, \qquad [7.4]$$

where $L(x_i, y_i)$ is the radiance value of the ith sample located at (x_i, y_i), and f is a filter function. Figure 7.36 shows a pixel at location (x, y) that has a pixel filter with extent

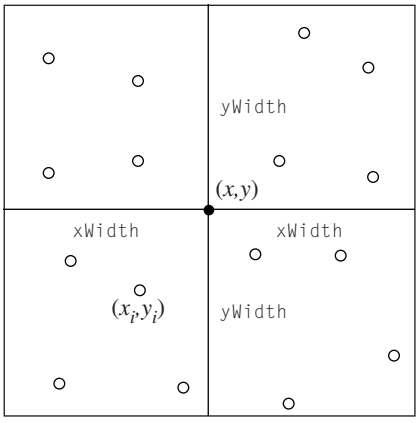
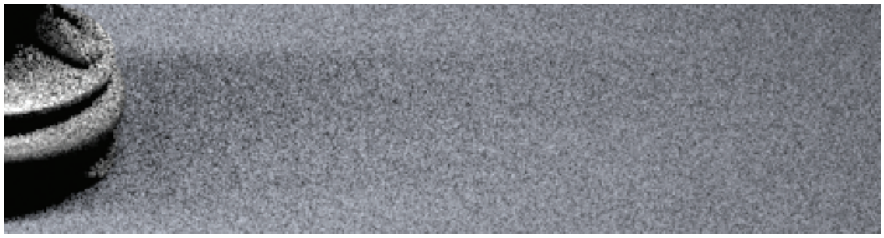
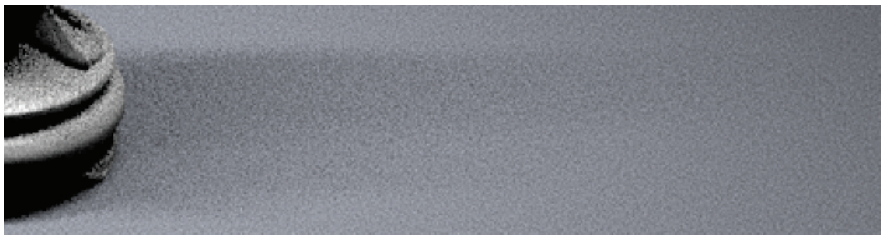


Figure 7.36: 2D Image Filtering. To compute a filtered pixel value for the pixel marked with a filled circle located at (x, y), all of the image samples inside the box around (x, y) with extent `xWidth` and `yWidth` need to be considered. Each of the image samples (x_i, y_i), denoted by open circles, is weighted by a 2D filter function, $f(x - x_i, y - y_i)$. The weighted average of all samples is the final pixel value.



(a)



(b)

Figure 7.37: The choice of pixel reconstruction filter interacts with the results from the sampling pattern in surprising ways. Here, a Mitchell filter has been used to reconstruct pixels in the soft shadows example with the `StratifiedSampler` (a) and the `LDSampler` (b). Note the significant difference and artifacts compared to the images in Figures 7.23 and 7.30. Here, we have used 16 image samples and 1 light sample per pixel.

`xWidth` in the x direction and `yWidth` in the y direction. All of the samples inside the box given by the filter extent may contribute to the pixel's value, depending on the filter function's value for $f(x - x_i, y - y_i)$.

The sinc filter is not an appropriate choice here: recall that the ideal sinc filter is prone to ringing when the underlying function has frequencies beyond the Nyquist limit (Gibbs phenomenon), meaning edges in the image have faint replicated copies of the edge in nearby pixels. Furthermore, the sinc filter has *infinite support*: it doesn't fall off to zero at a finite distance from its center, so all of the image samples would need to be filtered for each output pixel. In practice, there is no single best filter function. Choosing the best one for a particular scene takes a mixture of quantitative evaluation and qualitative judgment.

Another issue that influences the choice of image filter is that the reconstruction filter can interact with the sampling pattern in surprising ways. Recall the `LDSampler`: it generated an extremely well-distributed low-discrepancy pattern over the area of a single pixel, but samples in adjacent pixels were placed without regard for the samples in their neighbors. When used with a box filter, this sampling pattern works extremely well, but when a filter that both spans multiple pixels and isn't a constant value is used, it becomes less effective. Figure 7.37 shows this effect in practice. Using this filter with regular stratified samples makes much less of a difference. Given the remarkable effectiveness of patterns generated with the `LDSampler`, this is something of a quandary: the box filter is the last filter we'd like to use, yet using it instead of another filter substantially improves the results from

`LDSampler` 373
`StratifiedSampler` 349

the LDSampler. Given all of these issues, pbrt provides a variety of different filter function implementations.

7.7.1 FILTER FUNCTIONS

All filter implementations in pbrt are derived from an abstract Filter class, which provides the interface for the $f(x, y)$ functions used in filtering; see Equation (7.4). The Film class (described in the Section 7.8) stores a pointer to a Filter and uses it to filter the output before writing it to a file. Figure 7.38 shows comparisons of zoomed-in regions of images rendered using a variety of the filters from this section to reconstruct pixel values.

The Filter base class is defined in the files core/filter.h and core/filter.cpp.

```
⟨Filter Declarations⟩ ≡
   class Filter {
   public:
       ⟨Filter Interface 393⟩
       ⟨Filter Public Data 394⟩
   };
```

All filters define a width beyond which they have a value of zero; this width may be different in the x and y directions. The constructor takes these values and stores them along with their reciprocals, for use by the filter implementations. The filter's overall extent in each direction (its *support*) is twice the value of its corresponding width (Figure 7.39).

```
⟨Filter Interface⟩ ≡ 393
   Filter(float xw, float yw)
       : xWidth(xw), yWidth(yw), invXWidth(1.f/xw), invYWidth(1.f/yw) {
   }
```



(a) (b) (c)

Figure 7.38: The pixel reconstruction filter used to convert the image samples into pixel values can have a noticeable effect on the character of the final image. Here, we see blowups of a region of the brick wall in the Sponza atrium scene, filtered with (a) the box filter, (b) Gaussian, and (c) Mitchell–Netravali filter. Note that the Mitchell filter gives the sharpest image, while the Gaussian blurs it. The box is the least desirable, since it allows high-frequency aliasing to leak into the final image. (Note artifacts on the top edges of arches, for example.)

Film 403
Filter 393
LDSampler 373

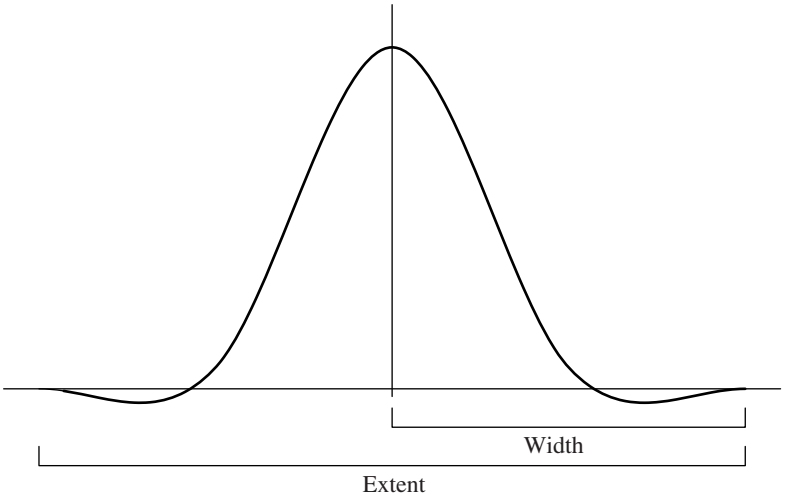


Figure 7.39: The extent of filters in `pbrt` is specified in terms of their width from the origin to its cutoff point. The support of a filter is its total non-zero extent, here equal to twice its width.

⟨*Filter Public Data*⟩ ≡ 393
```
const float xWidth, yWidth;
const float invXWidth, invYWidth;
```

The sole function that `Filter` implementations need to provide is `Evaluate()`. It takes x and y arguments, which give the position of the sample point relative to the center of the filter. The return value specifies the weight of the sample. Code elsewhere in the system will never call the filter function with points outside of the filter's extent, so filter implementations don't need to check for this case.

⟨*Filter Interface*⟩ +≡ 393
```
virtual float Evaluate(float x, float y) const = 0;
```

Box Filter

One of the most commonly used filters in graphics is the *box filter* (and, in fact, when filtering and reconstruction aren't addressed explicitly, the box filter is the *de facto* result). The box filter equally weights all samples within a square region of the image. Although computationally efficient, it's just about the worst filter possible. Recall from the discussion in Section 7.1 that the box filter allows high-frequency sample data to leak into the reconstructed values. This causes postaliasing—even if the original sample values were at a high enough frequency to avoid aliasing, errors are introduced by poor filtering.

Figure 7.40(a) shows a graph of the box filter, and Figure 7.41 shows the result of using the box filter to reconstruct two 1D functions. For the step function we used previously to illustrate the Gibbs phenomenon, the box does reasonably well. However, the results are much worse for a sinusoidal function that has increasing frequency along the x axis. Not only does the box filter do a poor job of reconstructing the function when the frequency

Filter 393

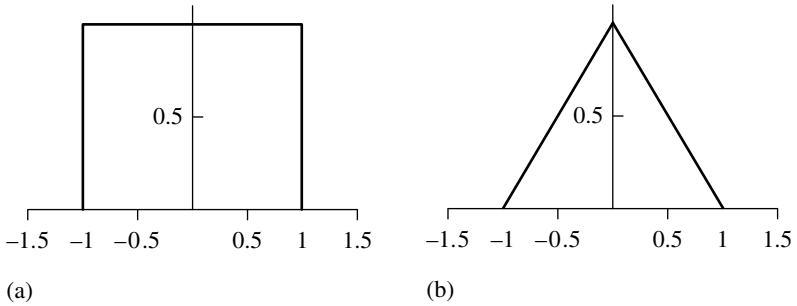


Figure 7.40: Graphs of the (a) box filter and (b) triangle filter. Although neither of these is a particularly good filter, they are both computationally efficient, easy to implement, and good baselines for evaluating other filters.

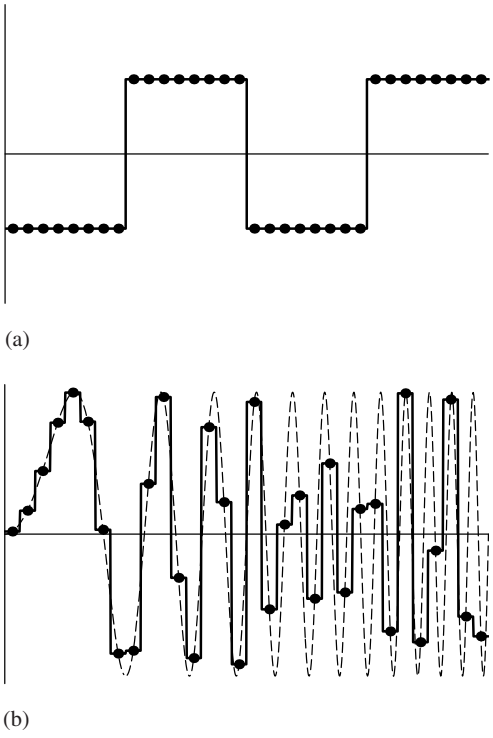


Figure 7.41: The box filter reconstructing (a) a step function and (b) a sinusoidal function with increasing frequency as x increases. This filter does well with the step function, as expected, but does an extremely poor job with the sinusoidal function.

is low, giving a discontinuous result even though the original function was smooth, but it also does an extremely poor job of reconstruction as the function's frequency approaches and passes the Nyquist limit.

⟨*Box Filter Declarations*⟩ ≡
```
class BoxFilter : public Filter {
public:
    BoxFilter(float xw, float yw) : Filter(xw, yw) { }
    float Evaluate(float x, float y) const;
};
```

Because the evaluation function won't be called with (x, y) values outside of the filter's extent, it can always return 1 for the filter function's value.

⟨*Box Filter Method Definitions*⟩ ≡
```
float BoxFilter::Evaluate(float x, float y) const {
    return 1.;
}
```

Triangle Filter

The triangle filter gives slightly better results than the box: samples at the filter center have a weight of one, and the weight falls off linearly to the square extent of the filter. See Figure 7.40(b) for a graph of the triangle filter.

⟨*Triangle Filter Declarations*⟩ ≡
```
class TriangleFilter : public Filter {
public:
    TriangleFilter(float xw, float yw) : Filter(xw, yw) { }
    float Evaluate(float x, float y) const;
};
```

Evaluating the triangle filter is simple: the implementation just computes a linear function based on the width of the filter in both the x and y directions.

⟨*Triangle Filter Method Definitions*⟩ ≡
```
float TriangleFilter::Evaluate(float x, float y) const {
    return max(0.f, xWidth - fabsf(x)) *
           max(0.f, yWidth - fabsf(y));
}
```

Gaussian Filter

Unlike the box and triangle filters, the Gaussian filter gives a reasonably good result in practice. This filter applies a Gaussian bump that is centered at the pixel and radially symmetric around it. The Gaussian's value at the end of its extent is subtracted from the filter value, in order to make the filter go to zero at its limit (Figure 7.42). The Gaussian does tend to cause slight blurring of the final image compared to some of the other filters, but this blurring can actually help mask any remaining aliasing in the image.

BoxFilter 396
Filter 393
Filter::xWidth 394
Filter::yWidth 394
TriangleFilter 396

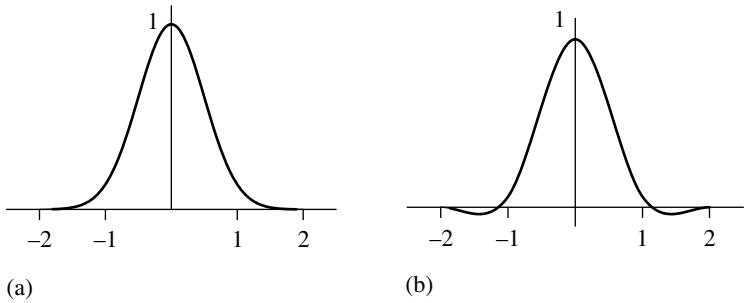


Figure 7.42: Graphs of (a) the Gaussian filter and (b) the Mitchell filter with $B = \frac{1}{3}$ and $C = \frac{1}{3}$, each with a width of two. The Gaussian gives images that tend to be a bit blurry, while the negative lobes of the Mitchell filter help to accentuate and sharpen edges in final images.

⟨*Gaussian Filter Declarations*⟩ ≡
```
class GaussianFilter : public Filter {
public:
    ⟨GaussianFilter Public Methods 397⟩
private:
    ⟨GaussianFilter Private Data 397⟩
    ⟨GaussianFilter Utility Functions 398⟩
};
```

The 1D Gaussian filter function of width w is

$$f(x) = \mathrm{e}^{-\alpha x^2} - \mathrm{e}^{-\alpha w^2},$$

where α controls the rate of falloff of the filter. Smaller values cause a slower falloff, giving a blurrier image. The second term here ensures that the Gaussian goes to zero at the end of its extent, rather than having an abrupt cliff. For efficiency, the constructor precomputes the constant term for $\mathrm{e}^{-\alpha w^2}$ in each direction.

⟨*GaussianFilter Public Methods*⟩ ≡ 397
```
GaussianFilter(float xw, float yw, float a)
    : Filter(xw, yw), alpha(a), expX(expf(-alpha * xWidth * xWidth)),
      expY(expf(-alpha * yWidth * yWidth)) { }
```

Filter 393
GaussianFilter 397
GaussianFilter::alpha 397
GaussianFilter::expX 397
GaussianFilter::expY 397

⟨*GaussianFilter Private Data*⟩ ≡ 397
```
const float alpha;
const float expX, expY;
```

Since a 2D Gaussian function is *separable* into the product of two 1D Gaussians, the implementation calls the `Gaussian()` function twice and multiplies the results.

⟨*Gaussian Filter Method Definitions*⟩ ≡
```
float GaussianFilter::Evaluate(float x, float y) const {
    return Gaussian(x, expX) * Gaussian(y, expY);
}
```

⟨*GaussianFilter Utility Functions*⟩ ≡ 397
```
float Gaussian(float d, float expv) const {
    return max(0.f, float(expf(-alpha * d * d) - expv));
}
```

Mitchell Filter

Filter design is notoriously difficult, mixing mathematical analysis and perceptual experiments. Mitchell and Netravali (1988) have developed a family of parameterized filter functions in order to be able to explore this space in a systematic manner. After analyzing test subjects' subjective responses to images filtered with a variety of parameter values, they developed a filter that tends to do a good job of trading off between *ringing* (phantom edges next to actual edges in the image) and *blurring* (excessively blurred results)—two common artifacts from poor reconstruction filters.

Note from the graph in Figure 7.42(b) that this filter function takes on negative values out by its edges; it has *negative lobes*. In practice these negative regions improve the sharpness of edges, giving crisper images (reduced blurring). If they become too large, however, ringing tends to start to enter the image. Also, because the final pixel values can therefore become negative, they will eventually need to be clamped to a legal output range.

Figure 7.43 shows this filter reconstructing the two test functions. It does extremely well with both of them: there is minimal ringing with the step function, and it does a very good job with the sinusoidal function, up until the point where the sampling rate isn't sufficient to capture the function's detail.

⟨*Mitchell Filter Declarations*⟩ ≡
```
class MitchellFilter : public Filter {
public:
    ⟨MitchellFilter Public Methods 398⟩
private:
    const float B, C;
};
```

The Mitchell filter has two parameters called B and C. Although any values can be used for these parameters, Mitchell and Netravali recommend that they lie along the line $B + 2C = 1$.

⟨*MitchellFilter Public Methods*⟩ ≡ 398
```
MitchellFilter(float b, float c, float xw, float yw)
    : Filter(xw, yw), B(b), C(c) {
}
```

The Mitchell-Netravali filter is the product of one-dimensional filter functions in the x and y directions and is therefore separable, like the Gaussian filter. (In fact, all of the provided filters in pbrt are separable.) Nevertheless, the `Filter::Evaluate()` interface

Filter 393
Filter::Evaluate() 394
GaussianFilter 397
GaussianFilter::expX 397
GaussianFilter::expY 397
MitchellFilter 398

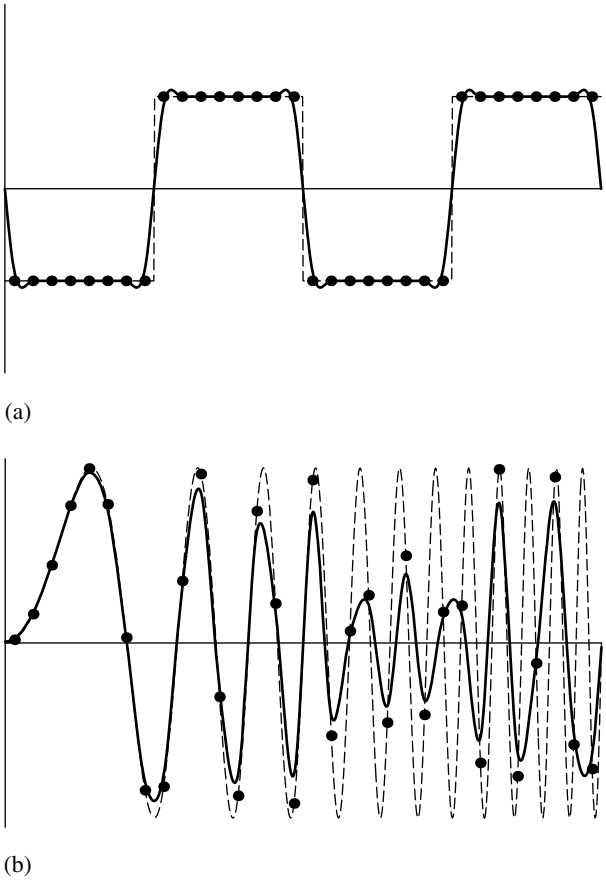


Figure 7.43: The Mitchell–Netravali Filter Used to Reconstruct the Example Functions. It does a good job with both of these functions, (a) introducing minimal ringing with the step function and (b) accurately representing the sinusoid until aliasing from undersampling starts to dominate.

does not enforce this requirement, giving more flexibility in implementing new filters in the future.

⟨*Mitchell Filter Method Definitions*⟩ ≡

```
float MitchellFilter::Evaluate(float x, float y) const {
    return Mitchell1D(x * invXWidth) * Mitchell1D(y * invYWidth);
}
```

Filter::invXWidth 394
Filter::invYWidth 394
MitchellFilter 398
MitchellFilter::Mitchell1D() 400

The 1D function used in the Mitchell filter is an even function defined over the range $[-2, 2]$. This function is made by joining a cubic polynomial defined over $[0, 1]$ with another cubic polynomial defined over $[1, 2]$. This combined polynomial is also reflected around the $x = 0$ plane to give the complete function. These polynomials are controlled by the B and C parameters and are chosen carefully to guarantee C^0 and C^1 continuity

at $x = 0$, $x = 1$, and $x = 2$. The polynomials are

$$f(x) = \frac{1}{6}\begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + (6 - 2B) & |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + (-12B - 48C)|x| + (8B + 24C) & 1 \le |x| < 2 \\ 0 & \text{otherwise.} \end{cases}$$

⟨*MitchellFilter Public Methods*⟩ +≡ 398

```
float Mitchell1D(float x) const {
    x = fabsf(2.f * x);
    if (x > 1.f)
        return ((-B - 6*C) * x*x*x + (6*B + 30*C) * x*x +
                (-12*B - 48*C) * x + (8*B + 24*C)) * (1.f/6.f);
    else
        return ((12 - 9*B - 6*C) * x*x*x +
                (-18 + 12*B + 6*C) * x*x +
                (6 - 2*B)) * (1.f/6.f);
}
```

Windowed Sinc Filter

Finally, the `LanczosSincFilter` class implements a filter based on the sinc function. In practice, the sinc filter is often multiplied by another function that goes to zero after some distance. This gives a filter function with finite extent, which is necessary for an implementation with reasonable performance. An additional parameter τ controls how many cycles the sinc function passes through before it is clamped to a value of zero, Figure 7.44 shows a graph of three cycles of the sinc function, along with a graph of the windowing function we use, which was developed by Lanczos. The Lanczos window is just the central lobe of the sinc function, scaled to cover the τ cycles:

$$w(x) = \frac{\sin \pi x/\tau}{\pi x/\tau}.$$

Figure 7.44 also shows the filter that we will implement here, which is the product of the sinc function and the windowing function.

Figure 7.45 shows the windowed sinc's reconstruction results for uniform 1D samples. Thanks to the windowing, the reconstructed step function exhibits far less ringing than the reconstruction using the infinite-extent sinc function (compare to Figure 7.11). The windowed sinc filter also does extremely well at reconstructing the sinusoidal function until prealiasing begins.

⟨*Sinc Filter Declarations*⟩ ≡

```
class LanczosSincFilter : public Filter {
public:
    ⟨LanczosSincFilter Public Methods 402⟩
private:
    const float tau;
};
```

Filter 393
LanczosSincFilter 400

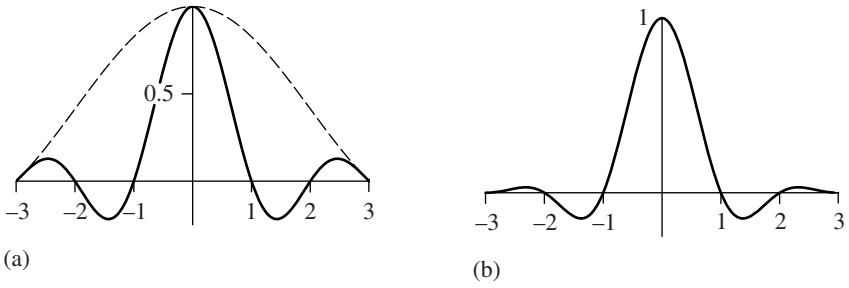


Figure 7.44: Graphs of the Sinc Filter. (a) The sinc function, truncated after three cycles (solid line) and the Lanczos windowing function (dashed line). (b) The product of these two functions, as implemented in the `LanczosSincFilter`.

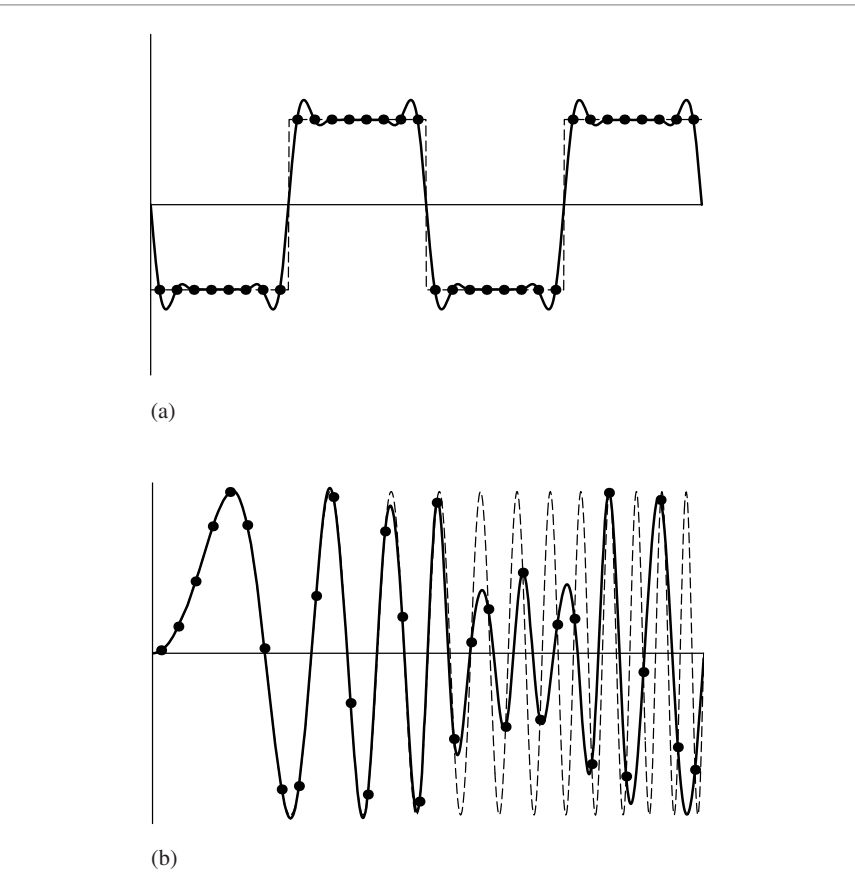


Figure 7.45: Results of Using the Windowed Sinc Filter to Reconstruct the Example Functions. Here, $\tau = 3$. (a) Like the infinite sinc, it suffers from ringing with the step function, although there is much less ringing in the windowed version. (b) The filter does quite well with the sinusoid, however.

⟨*LanczosSincFilter Public Methods*⟩ ≡ 400
```
LanczosSincFilter(float xw, float yw, float t)
    : Filter(xw, yw), tau(t) { }
```

⟨*Sinc Filter Method Definitions*⟩ ≡
```
float LanczosSincFilter::Evaluate(float x, float y) const {
    return Sinc1D(x * invXWidth) * Sinc1D(y * invYWidth);
}
```

The implementation computes the value of the sinc function and then multiplies it by the value of the Lanczos windowing function.

⟨*LanczosSincFilter Public Methods*⟩ +≡ 400
```
float Sinc1D(float x) const {
    x = fabsf(x);
    if (x < 1e-5) return 1.f;
    if (x > 1.)   return 0.f;
    x *= M_PI;
    float sinc = sinf(x * tau) / (x * tau);
    float lanczos = sinf(x) / x;
    return sinc * lanczos;
}
```

7.8 FILM AND THE IMAGING PIPELINE

The type of film or sensor in a camera has a dramatic effect on the way that incident light is eventually transformed into colors in an image. In pbrt, the Film class models the sensing device in the simulated camera. After the radiance is found for each camera ray, a Film implementation determines the sample's contribution to the nearby pixels and updates its representation of the image. When the main rendering loop exits, the Film typically writes the final image to a file on disk.

This section only provides a single Film implementation. It applies the pixel reconstruction equation to compute final pixel values and writes the image to disk with floating-point color values. For a physically based renderer, creating images in a floating-point format provides more flexibility in how the output can be used than if a typical image format with 8-bit unsigned integer values is used; floating-point formats avoid the substantial loss of information that comes from image quantization to 8-bit image formats.

In order to display such images on modern display devices, however, it is necessary to map these floating-point pixel values to discrete values for display. For example, computer monitors generally expect the color of each pixel to be described by an RGB color triple, not an arbitrary spectral power distribution. Spectra described by general basis function coefficients must therefore be converted to an RGB representation before they can be displayed. A related problem is that displays have a substantially smaller range of displayable radiance values than the range present in many real-world scenes. Therefore, the pixel values must be mapped to the displayable range in a way that causes the final displayed image to appear as close as possible to the way it would appear on an ideal

Film 403
Filter 393
Filter::invXWidth 394
Filter::invYWidth 394
LanczosSincFilter 400
LanczosSincFilter::Sinc1D() 402
M_PI 1002

display device without this limitation. These issues are addressed by research into *tone mapping*; the "Further Reading" section has more information about this topic.

7.8.1 FILM INTERFACE

The `Film` base class, defined in `core/film.h` and `core/film.cpp`, defines the abstract interface for `Film` implementations:

⟨*Film Declarations*⟩ ≡
```
class Film {
public:
    ⟨Film Interface 403⟩
    ⟨Film Public Data 403⟩
};
```

The `Film` constructor must be given the overall resolution of the image in the x and y directions; these are stored in the public member variables `Film::xResolution` and `Film::yResolution`. The `Cameras` in Chapter 6 need these values to compute some of the camera-related transformations, such as the raster-to-camera-space transformations.

⟨*Film Interface*⟩ ≡ 403
```
Film(int xres, int yres)
    : xResolution(xres), yResolution(yres) { }
```

⟨*Film Public Data*⟩ ≡ 403
```
const int xResolution, yResolution;
```

There are two methods for providing data to the film. The first is driven by `Samples` generated by the `Sampler`; for each such sample, `Film::AddSample()` is called. It takes a sample and corresponding radiance value, applies the reconstruction filter, and and updates the stored image. The expectation with this method is that the sampling density around a pixel is not related to the pixel's final value. (In other words, the final pixel value is effectively a weighted average of the nearby samples.) Under this assumption, the pixel filtering equation can be used to compute the final values.

⟨*Film Interface*⟩ +≡ 403
```
virtual void AddSample(const CameraSample &sample,
                       const Spectrum &L) = 0;
```

The second method for providing data to the film is `Splat()`. Splatting similarly updates pixel values, but rather than computing the final pixel value as a weighted average, splats are simply summed. Thus, the more splats that are around a given pixel, the brighter the pixel will be. This method is used by light transport algorithms like the one in the `MetropolisRenderer`, where more samples are generated in areas where the image is brighter. It would also be used by bidirectional light transport algorithms that followed paths starting from lights and then projected illuminated points onto the film.

⟨*Film Interface*⟩ +≡ 403
```
virtual void Splat(const CameraSample &sample, const Spectrum &L) = 0;
```

The `Film` is responsible for determining the range of integer pixel values that the `Sampler` is responsible for generating samples for. While this range would be from (0, 0) to

Camera 302
CameraSample 342
Film 403
Film::AddSample() 403
Film::xResolution 403
Film::yResolution 403
MetropolisRenderer 852
Sample 343
Sampler 340
Spectrum 263

(xResolution − 1, yResolution − 1) for a simple film implementation, in general it is necessary to sample the image plane at locations slightly beyond the edges of the final image due to the finite extent of pixel reconstruction filters. This range is returned by the `GetSampleExtent()` method.

⟨*Film Interface*⟩ +≡ 403
```
virtual void GetSampleExtent(int *xstart, int *xend,
                             int *ystart, int *yend) const = 0;
```

`GetPixelExtent()` provides the range of pixels in the actual image.

⟨*Film Interface*⟩ +≡ 403
```
virtual void GetPixelExtent(int *xstart, int *xend,
                            int *ystart, int *yend) const = 0;
```

Some `Film` implementations find it useful to be notified that a region of the pixels has recently been updated. In particular, an implementation that opened a window to show the image as it was being rendered could use this information to trigger an update of a subregion of the window after its samples had been generated. Here we provide a default empty implementation of this method so that `Film` implementations that don't need this information don't need to implement it themselves.

⟨*Film Method Definitions*⟩ ≡
```
void Film::UpdateDisplay(int x0, int y0, int x1, int y1,
                         float splatScale) {
}
```

After the main rendering loop exits, the `SamplerRenderer::Render()` method calls the `Film::WriteImage()` method, which allows the film to do any processing necessary to generate the final image and display it or store it in a file. Like `UpdateDisplay()`, it takes a scale factor for the samples provided to the `Splat()` method.

⟨*Film Interface*⟩ +≡ 403
```
virtual void WriteImage(float splatScale = 1.f) = 0;
```

7.8.2 IMAGEFILM

The `ImageFilm` is a `Film` implementation that filters image sample values with a given reconstruction filter and writes the resulting image to disk. It is implemented in the files `film/image.h` and `film/image.cpp`.

⟨*ImageFilm Declarations*⟩ ≡
```
class ImageFilm : public Film {
public:
    ⟨ImageFilm Public Methods⟩
private:
    ⟨ImageFilm Private Data 405⟩
};
```

The `ImageFilm` constructor takes a number of extra parameters beyond the overall image resolution, including a filter function, a crop window that specifies a subrectangle of the pixels to be rendered, and the filename for the output image. On some systems, the

Film 403
Film::WriteImage() 404
ImageFilm 404
SamplerRenderer::Render() 27

ImageFilm can be configured to open a window and show the image as it's being rendered; the openWindow parameter to the constructor controls this. (In the interest of brevity, we won't include the code related to this in the book here, however.)

⟨*ImageFilm Method Definitions*⟩ ≡
```
ImageFilm::ImageFilm(int xres, int yres, Filter *filt, const float crop[4],
                     const string &fn, bool openWindow)
    : Film(xres, yres) {
    filter = filt;
    memcpy(cropWindow, crop, 4 * sizeof(float));
    filename = fn;
    ⟨Compute film image extent 406⟩
    ⟨Allocate film image storage 407⟩
    ⟨Precompute filter weight table 407⟩
    ⟨Possibly open window for image display⟩
}
```

⟨*ImageFilm Private Data*⟩ ≡ 404
```
Filter *filter;
float cropWindow[4];
string filename;
```

In conjunction with the overall image resolution, the crop window gives the extent of pixels that need to be actually stored and written out. Crop windows are useful for debugging or for breaking a large image into small pieces that can be rendered on different computers and reassembled later. The crop window is specified in NDC space, with each coordinate ranging from zero to one (Figure 7.46). ImageFilm::xPixelStart and ImageFilm::yPixelStart store the pixel position of the upper-left corner of the crop window, and ImageFilm::xPixelCount and ImageFilm::yPixelCount give the total number of pixels in each direction. Their values are easily computed from the overall image resolution and the crop window, although the calculations must be done carefully such that if

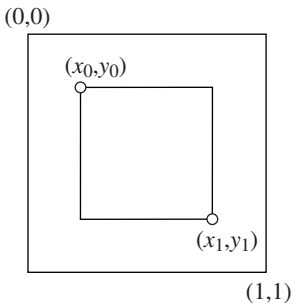


Figure 7.46: The image crop window specifies a subset of the image to be rendered. It is given in NDC space, with coordinates ranging from (0, 0) to (1, 1). The ImageFilm class only allocates space for and stores pixel values in the region inside the crop window.

Film 403
Filter 393
ImageFilm 404
ImageFilm::cropWindow 405
ImageFilm::filename 405
ImageFilm::filter 405
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406

an image is rendered in pieces with abutting crop windows, each final pixel will be present in only one of the subimages.

⟨*Compute film image extent*⟩ ≡ 405
```
xPixelStart = Ceil2Int(xResolution * cropWindow[0]);
xPixelCount = max(1, Ceil2Int(xResolution * cropWindow[1]) - xPixelStart);
yPixelStart = Ceil2Int(yResolution * cropWindow[2]);
yPixelCount = max(1, Ceil2Int(yResolution * cropWindow[3]) - yPixelStart);
```

⟨*ImageFilm Private Data*⟩ +≡ 404
```
int xPixelStart, yPixelStart, xPixelCount, yPixelCount;
```

Given the pixel resolution of the (possibly cropped) image, the constructor allocates an array of `Pixel` structures, one for each pixel. The running weighted sums of pixel radiance values are stored in the `Lxyz` member variable, and `weightSum` holds the sum of filter weight values for the sample contributions to the pixel. `splatXYZ` holds the (unweighted) sum of sample splats. The `pad` member is unused; its sole purpose is to ensure that the `Pixel` structure is 32 bytes large, rather than 28 as it would be otherwise. This ensures that a `Pixel` won't straddle a cache line, so that no more than one cache miss will be incurred when a `Pixel` is accessed (as long as the first `Pixel` in the array is allocated at the start of a cache line).

The `ImageFilm` uses XYZ color values to store pixel colors (Section 5.2.1). Two natural alternatives would be to use `Spectrum` values, or to store RGB color. Here, it isn't worthwhile to store complete `Spectrum` values, even when doing full spectral rendering. Because the final colors written to the output file don't include the full set of `Spectrum` samples, converting to a tristimulus value here doesn't represent a loss of information versus storing `Spectrum`s and converting to a tristimulus value on image output. Not storing complete `Spectrum` values in this case can save a substantial amount of memory if the `Spectrum` has a large number of samples.

We have chosen to use XYZ color rather than RGB to emphasize that XYZ is a display-independent representation of color, while RGB requires assuming a particular set of display response curves (Section 5.2.2). (In the end, we will, however, have to convert to RGB, since few image file formats store XYZ color.)

⟨*ImageFilm Private Data*⟩ +≡ 404
```
struct Pixel {
    Pixel() {
        for (int i = 0; i < 3; ++i) Lxyz[i] = splatXYZ[i] = 0.f;
        weightSum = 0.f;
    }
    float Lxyz[3];
    float weightSum;
    float splatXYZ[3];
    float pad;
};
BlockedArray<Pixel> *pixels;
```

BlockedArray 1017
Ceil2Int() 1002
Film::xResolution 403
Film::yResolution 403
ImageFilm::cropWindow 405
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel 406
Pixel::weightSum 406
Spectrum 263

Because small rectangular blocks of pixels are typically accessed for each image sample, the `ImageFilm` uses a `BlockedArray` to store the pixels, which helps reduce the number of cache misses as samples arrive and pixel values are updated. The `BlockedArray` also ensures that the dynamically allocated memory for the array is cache aligned.

⟨*Allocate film image storage*⟩ ≡ 405
```
pixels = new BlockedArray<Pixel>(xPixelCount, yPixelCount);
```

With typical filter settings, every image sample may contribute to 16 or more pixels in the final image. Particularly for simple scenes, where relatively little time is spent on ray intersection testing and shading computations, the time spent updating the image for each sample can be significant. Therefore, the `ImageFilm` precomputes a table of filter values so that the `Film::AddSample()` method can avoid the expense of virtual function calls to the `Filter::Evaluate()` method as well as the expense of evaluating the filter and can instead use values from the table for filtering. The error introduced by not evaluating the filter at each sample's precise location isn't noticeable in practice.

The implementation here makes the reasonable assumption that the filter is defined such that $f(x, y) = f(|x|, |y|)$, so the table needs to hold values for only the positive quadrant of filter offsets. This assumption is true for all of the `Filters` currently available in `pbrt` and is true for most filters used in practice. This makes the table one-fourth the size and improves the coherence of memory accesses, leading to better cache performance.[8]

⟨*Precompute filter weight table*⟩ ≡ 405
```
#define FILTER_TABLE_SIZE 16
filterTable = new float[FILTER_TABLE_SIZE * FILTER_TABLE_SIZE];
float *ftp = filterTable;
for (int y = 0; y < FILTER_TABLE_SIZE; ++y) {
    float fy = ((float)y + .5f) *
               filter->yWidth / FILTER_TABLE_SIZE;
    for (int x = 0; x < FILTER_TABLE_SIZE; ++x) {
        float fx = ((float)x + .5f) *
                   filter->xWidth / FILTER_TABLE_SIZE;
        *ftp++ = filter->Evaluate(fx, fy);
    }
}
```

⟨*ImageFilm Private Data*⟩ +≡ 404
```
float *filterTable;
```

To understand the operation of `ImageFilm::AddSample()`, first recall the pixel filtering equation:

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}.$$

BlockedArray 1017
Film::AddSample() 403
Filter 393
Filter::Evaluate() 394
Filter::xWidth 394
Filter::yWidth 394
FILTER_TABLE_SIZE 407
ImageFilm 404
ImageFilm::AddSample() 408
ImageFilm::filter 405
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::yPixelCount 406
Pixel 406

8 The implementation here could further take advantage of the fact that all filters currently in `pbrt` are separable, only allocating a 1D table. However, to more easily allow different filter functions to be added, we don't assume separability here.

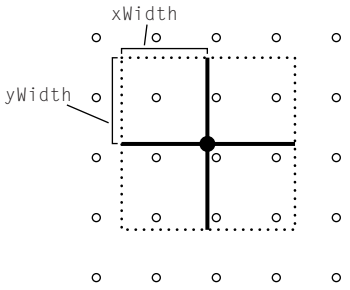


Figure 7.47: Given an image sample at some position on the image plane (solid dot), it is necessary to determine which pixel values (empty dots) are affected by the sample's contribution. This is done by taking the offsets in the x and y directions according to the pixel reconstruction filter's width (solid lines) and finding the pixels inside this region.

It computes each pixel's value $I(x, y)$ as the weighted sum of nearby samples' radiance values, using a filter function f to compute the weights. Because all of the `Filters` in `pbrt` have finite extent, this method starts by computing which pixels will be affected by the current sample. Then, turning the pixel filtering equation inside out, it updates two running sums for each pixel (x, y) that is affected by the sample. One sum accumulates the numerator of the pixel filtering equation and the other accumulates the denominator. When all of the samples have been processed, the final pixel values are computed by performing the division.

```
⟨ImageFilm Method Definitions⟩ +≡
  void ImageFilm::AddSample(const CameraSample &sample,
                            const Spectrum &L) {
      ⟨Compute sample's raster extent 408⟩
      ⟨Loop over filter support and add sample to pixel arrays 409⟩
  }
```

To find which pixels a sample potentially contributes to, `Film::AddSample()` converts the continuous sample coordinates to discrete coordinates by subtracting 0.5 from x and y. It then offsets this value by the filter width in each direction (Figure 7.47) and takes the ceiling of the minimum coordinates and the floor of the maximum, since pixels outside the bound of the extent are guaranteed to be unaffected by the sample.

```
⟨Compute sample's raster extent⟩ ≡ 408
  float dimageX = sample.imageX - 0.5f;
  float dimageY = sample.imageY - 0.5f;
  int x0 = Ceil2Int (dimageX - filter->xWidth);
  int x1 = Floor2Int(dimageX + filter->xWidth);
  int y0 = Ceil2Int (dimageY - filter->yWidth);
  int y1 = Floor2Int(dimageY + filter->yWidth);
  x0 = max(x0, xPixelStart);
  x1 = min(x1, xPixelStart + xPixelCount - 1);
```

CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
Ceil2Int() 1002
Film::AddSample() 403
Filter 393
Filter::xWidth 394
Filter::yWidth 394
Floor2Int() 1002
ImageFilm 404
ImageFilm::filter 405
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Spectrum 263

```
y0 = max(y0, yPixelStart);
y1 = min(y1, yPixelStart + yPixelCount - 1);
if ((x1-x0) < 0 || (y1-y0) < 0)
    return;
```

Given the extent of pixels that are affected by this sample—(x0,y0) to (x1,y1), inclusive—the method can now loop over all of those pixels and then filter the sample value appropriately.

Before the loop starts, it checks to see if the filter is greater than one pixel wide. If this is true, then sample values from more than one SamplerRendererTask from the SamplerRenderer may affect a particular pixel's value. (Samples near the boundary of their image tiles will affect pixels in other image tiles due to the filter's extent.) In this case, we must ensure that multiple threads entering this method at the same time correctly coordinate their updates of pixel values. On the other hand, if the filter is a pixel wide (or less), then we know that only a single SamplerRendererTask will be updating any pixel's value, and that therefore no synchronization is needed for pixel updates here.[9]

⟨*Loop over filter support and add sample to pixel arrays*⟩ ≡ 408

```
float xyz[3];
L.ToXYZ(xyz);
⟨Precompute x and y filter table offsets 410⟩
bool syncNeeded = (filter->xWidth > 0.5f || filter->yWidth > 0.5f);
for (int y = y0; y <= y1; ++y) {
    for (int x = x0; x <= x1; ++x) {
        ⟨Evaluate filter value at (x, y) pixel 410⟩
        ⟨Update pixel values with filtered sample contribution 410⟩
    }
}
```

Each discrete integer pixel (x, y) has an instance of the filter function centered around it. To compute the filter weight for a particular sample, it's necessary to find the offset from the pixel to the sample's position in discrete coordinates and evaluate the filter function. If we were evaluating the filter explicitly, the appropriate computation would be

```
filterWt = filter->Evaluate(x - dimageX, y - dimageY);
```

Instead, the implementation retrieves the appropriate filter weight from the table.

To find the filter weight for a pixel (x', y') given the sample position (x, y), this routine computes the offset $(x' - x, y' - y)$ and converts it into coordinates for the filter weights lookup table. This can be done directly by dividing each component of the sample offset by the filter width in that direction, giving a value between zero and one, and then multiplying by the table size. This process can be further optimized by noting that along each row of pixels in the x direction, the difference in y, and thus the y offset into the filter table, is constant. Analogously, for each column of pixels, the x offset is constant.

Filter::xWidth 394
Filter::yWidth 394
ImageFilm 404
ImageFilm::filter 405
Renderer 24
SamplerRenderer 25
SamplerRendererTask 29
Spectrum::ToXYZ() 272

9 Note that this represents a potentially undesirable coupling between the parallelization strategy in the SamplerRenderer and the ImageFilm here; for another Renderer that used a different parallelization scheme such that multiple tasks updated the same pixel value, this expectation would be incorrect.

Therefore, before looping over the pixels here it's possible to precompute these indices, saving repeated work in the loop.

⟨*Precompute x and y filter table offsets*⟩ ≡ 409
```
int *ifx = ALLOCA(int, x1 - x0 + 1);
for (int x = x0; x <= x1; ++x) {
    float fx = fabsf((x - dimageX) *
                     filter->invXWidth * FILTER_TABLE_SIZE);
    ifx[x-x0] = min(Floor2Int(fx), FILTER_TABLE_SIZE-1);
}
int *ify = ALLOCA(int, y1 - y0 + 1);
for (int y = y0; y <= y1; ++y) {
    float fy = fabsf((y - dimageY) *
                     filter->invYWidth * FILTER_TABLE_SIZE);
    ify[y-y0] = min(Floor2Int(fy), FILTER_TABLE_SIZE-1);
}
```

Now at each pixel, the x and y offsets into the filter table can be found for the pixel, leading to the offset into the array and thus the filter value.

⟨*Evaluate filter value at* (x, y) *pixel*⟩ ≡ 409
```
int offset = ify[y-y0]*FILTER_TABLE_SIZE + ifx[x-x0];
float filterWt = filterTable[offset];
```

For each affected pixel, we need to add the weighted XYZ color and the filter weight. If no synchronization is needed, standard addition suffices.

⟨*Update pixel values with filtered sample contribution*⟩ ≡ 409
```
Pixel &pixel = (*pixels)(x - xPixelStart, y - yPixelStart);
if (!syncNeeded) {
    pixel.Lxyz[0] += filterWt * xyz[0];
    pixel.Lxyz[1] += filterWt * xyz[1];
    pixel.Lxyz[2] += filterWt * xyz[2];
    pixel.weightSum += filterWt;
}
else {
    ⟨Safely update Lxyz and weightSum even with concurrency 411⟩
}
```

Otherwise, we have to handle the case of multiple threads potentially trying to update this pixel's values simultaneously. A number of options are possible. We could require that a mutex be acquired before updating any pixel values, but as the number of simultaneously executing threads increased, this mutex could become a point of contention: if any thread was updating any part of the image, none of the others would be able to update even a completely different part of the image.

This problem in turn could be addressed by using multiple mutexes, where each mutex was responsible for controlling access to a subset of the image (a scanline, a tile of small size, etc). In this case, we'd just need to acquire the appropriate set of mutexes, update,

ALLOCA() 1009
Filter::invXWidth 394
Filter::invYWidth 394
FILTER_TABLE_SIZE 407
Floor2Int() 1002
ImageFilm::filterTable 407
ImageFilm::pixels 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelStart 406
Pixel 406

and return. This approach may instead suffer from the overhead of multiple mutex operations, however.

Therefore, here we use atomic processor operations to safely update each of the pixel values; the processor will ensure that each update is computed correctly. Another nice property of this approach is that under the common case when no two threads are trying to modify the same pixel values, the overhead of these atomic operations is reasonable; with many hardware architectures, only as multiple threads are concurrently trying to modify the same location does the cost of atomics increase.

⟨*Safely update* `Lxyz` *and* `weightSum` *even with concurrency*⟩ ≡ 410
```
AtomicAdd(&pixel.Lxyz[0], filterWt * xyz[0]);
AtomicAdd(&pixel.Lxyz[1], filterWt * xyz[1]);
AtomicAdd(&pixel.Lxyz[2], filterWt * xyz[2]);
AtomicAdd(&pixel.weightSum, filterWt);
```

The implementation of the `Splat()` method here just has to compute which pixel the sample maps to and safely add the value. It always uses atomic operations under the expectation that, in general, it's likely that multiple threads may need to splat values to the same pixel. The implementation here effectively uses a pixel-wide box filter to filter the samples; Exercise 7.3 at the end of the chapter discusses how to address this shortcoming.

⟨*ImageFilm Method Definitions*⟩ +≡
```
void ImageFilm::Splat(const CameraSample &sample, const Spectrum &L) {
    float xyz[3];
    L.ToXYZ(xyz);
    int x = Floor2Int(sample.imageX), y = Floor2Int(sample.imageY);
    if (x < xPixelStart || x - xPixelStart >= xPixelCount ||
        y < yPixelStart || y - yPixelStart >= yPixelCount) return;
    Pixel &pixel = (*pixels)(x - xPixelStart, y - yPixelStart);
    AtomicAdd(&pixel.splatXYZ[0], xyz[0]);
    AtomicAdd(&pixel.splatXYZ[1], xyz[1]);
    AtomicAdd(&pixel.splatXYZ[2], xyz[2]);
}
```

AtomicAdd() 1038
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
Filter::xWidth 394
Filter::yWidth 394
Floor2Int() 1002
ImageFilm 404
ImageFilm::filter 405
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel 406
Pixel::Lxyz 406
Pixel::splatXYZ 406
Pixel::weightSum 406
Sampler 340
Spectrum 263
Spectrum::ToXYZ() 272

Because the pixel reconstruction filter spans a number of pixels, the `Sampler` must generate image samples a bit outside of the range of pixels that will actually be output. This way, even pixels at the boundary of the image will have an equal density of samples around them in all directions and won't be biased with only values from toward the interior of the image. This is also important when rendering images in pieces with crop windows, since it eliminates artifacts at the edges of the subimages.

⟨*ImageFilm Method Definitions*⟩ +≡
```
void ImageFilm::GetSampleExtent(int *xstart, int *xend,
                                int *ystart, int *yend) const {
    *xstart = Floor2Int(xPixelStart + 0.5f - filter->xWidth);
    *xend   = Floor2Int(xPixelStart + 0.5f + xPixelCount  +
                        filter->xWidth);
```

```
    *ystart = Floor2Int(yPixelStart + 0.5f - filter->yWidth);
    *yend   = Floor2Int(yPixelStart + 0.5f + yPixelCount +
                        filter->yWidth);
}
```

⟨*ImageFilm Method Definitions*⟩ +≡

```
void ImageFilm::GetPixelExtent(int *xstart, int *xend,
                               int *ystart, int *yend) const {
    *xstart = xPixelStart;
    *xend   = xPixelStart + xPixelCount;
    *ystart = yPixelStart;
    *yend   = yPixelStart + yPixelCount;
}
```

Image Output

After the main rendering loop finishes, `SamplerRenderer::Render()` calls the `Film::WriteImage()` method to store the final image in a file.

⟨*ImageFilm Method Definitions*⟩ +≡

```
void ImageFilm::WriteImage(float splatScale) {
    ⟨Convert image to RGB and compute final pixel values 412⟩
    ⟨Write RGB image 413⟩
    ⟨Release temporary image memory 413⟩
}
```

This method starts by allocating an array to store the RGB pixel values.

⟨*Convert image to RGB and compute final pixel values*⟩ ≡ 412

```
int nPix = xPixelCount * yPixelCount;
float *rgb = new float[3*nPix];
int offset = 0;
for (int y = 0; y < yPixelCount; ++y) {
    for (int x = 0; x < xPixelCount; ++x) {
        ⟨Convert pixel XYZ color to RGB 412⟩
        ⟨Normalize pixel with weight sum 413⟩
        ⟨Add splat value at pixel 413⟩
        ++offset;
    }
}
```

Given information about the response characteristics of the display device being used, the pixel values can be converted to device-dependent RGB values from the device-independent XYZ tristimulus values. This conversion is another change of spectral basis, where the new basis is determined by the spectral response curves of the red, green, and blue elements of the display device. Here, weights to convert from XYZ to the device RGB based on the HDTV standard are used. This is a good match for most modern display devices.

⟨*Convert pixel XYZ color to RGB*⟩ ≡ 412

```
XYZToRGB((*pixels)(x, y).Lxyz, &rgb[3*offset]);
```

Film::WriteImage() 404
ImageFilm 404
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel::Lxyz 406
SamplerRenderer::Render() 27
XYZToRGB() 273

As the RGB output values are being initialized, their final values from the pixel filtering equation are computed by dividing each pixel sample value by `Pixel::weightSum`. This conversion can lead to RGB values where some components are negative; these are *out-of-gamut* colors that can't be represented with the chosen display primaries. Various approaches have been suggested to deal with this issue, ranging from clamping to zero, offsetting all components to lie within the gamut, or even performing a global optimization based on all of the pixels in the image. In addition to out-of-gamut colors, reconstructed pixels may also end up with negative values due to negative lobes in the reconstruction filter function. In order to handle both of these cases, their components are clamped to zero here.

⟨*Normalize pixel with weight sum*⟩ ≡ 412
```
float weightSum = (*pixels)(x, y).weightSum;
if (weightSum != 0.f) {
    float invWt = 1.f / weightSum;
    rgb[3*offset  ] = max(0.f, rgb[3*offset  ] * invWt);
    rgb[3*offset+1] = max(0.f, rgb[3*offset+1] * invWt);
    rgb[3*offset+2] = max(0.f, rgb[3*offset+2] * invWt);
}
```

It's also necessary to add in the splatted values for this pixel to the final value.

⟨*Add splat value at pixel*⟩ ≡ 412
```
float splatRGB[3];
XYZToRGB((*pixels)(x, y).splatXYZ, splatRGB);
rgb[3*offset  ] += splatScale * splatRGB[0];
rgb[3*offset+1] += splatScale * splatRGB[1];
rgb[3*offset+2] += splatScale * splatRGB[2];
```

The `WriteImage()` function, defined in Section A.2 in Appendix A, handles the details of writing the image to a file:

⟨*Write RGB image*⟩ ≡ 412
```
::WriteImage(filename, rgb, NULL, xPixelCount, yPixelCount,
             xResolution, yResolution, xPixelStart, yPixelStart);
```

After saving the image, the working memory is freed.

⟨*Release temporary image memory*⟩ ≡ 412
```
delete[] rgb;
```

Film::xResolution 403
Film::yResolution 403
ImageFilm::filename 405
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel::splatXYZ 406
Pixel::weightSum 406
WriteImage() 1004
XYZToRGB() 273

FURTHER READING

One of the best books on signal processing, sampling, reconstruction, and the Fourier transform is Bracewell's *The Fourier Transform and Its Applications* (Bracewell 2000). Glassner's *Principles of Digital Image Synthesis* (Glassner 1995) has a series of chapters on the theory and application of uniform and nonuniform sampling and reconstruction to computer graphics. For an extensive survey of the history of and techniques for interpolation of sampled data, including the sampling theorem, see Meijering (2002).

Unser (2000) also surveyed recent developments in sampling and reconstruction theory including the recent move away from focusing purely on band-limited functions.

Crow (1977) first identified aliasing as a major source of artifacts in computer-generated images. Using nonuniform sampling to turn aliasing into noise was introduced by Cook (1986) and Dippé and Wold (1985); their work was based on experiments by Yellot (1983), who investigated the distribution of photoreceptors in the eyes of monkeys. Dippé and Wold also first introduced the pixel filtering equation to graphics and developed a Poisson sample pattern with a minimum distance between samples. Lee, Redner, and Uselton (1985) developed a technique for adaptive sampling based on statistical tests that computed images to a given error tolerance.

Heckbert (1990a) wrote an article that explains possible pitfalls when using floating-point coordinates for pixels and develops the conventions used here.

Mitchell investigated sampling patterns for ray tracing extensively. His 1987 and 1991 SIGGRAPH papers on this topic have many key insights, and the best-candidate approach described in this chapter is based on the latter paper (Mitchell 1987, 1991). The general interface for `Samplers` in `pbrt` is based on an approach he has used in his implementations (Mitchell 1996a). Lagae and Dutré (2008c) surveyed the state of the art in generating Poission disk sample patterns; a number of recent approaches are substantially more efficient than the best-candidate dart-throwing method used in this chapter. See in particular the papers by Jones (2005), Dunbar and Humphreys (2006), and Wei (2008).

Shirley (1991) first introduced the use of discrepancy to evaluate the quality of sample patterns in computer graphics. This work was built upon by Mitchell (1992), Dobkin and Mitchell (1993), and Dobkin, Eppstein, and Mitchell (1996). One important observation in Dobkin et al.'s paper is that the box discrepancy measure used in this chapter and in other work that applies discrepancy to pixel sampling patterns isn't particularly appropriate for measuring a sampling pattern's accuracy at randomly oriented edges through a pixel, and that a discrepancy measure based on random edges should be used instead. This observation explains why many of the theoretically good low-discrepancy patterns do not perform as well as expected when used for image sampling.

Mitchell's first paper on discrepancy introduced the idea of using deterministic low-discrepancy sequences for sampling, removing all randomness in the interest of lower discrepancy (Mitchell 1992). Such *quasi-random* sequences are the basis of quasi–Monte Carlo methods, which will be described in Chapter 14. The seminal book on quasi-random sampling and algorithms for generating low-discrepancy patterns was written by Niederreiter (1992). Using random digit permutations with the Halton sequence was suggested by Ökten and Shah (2008).

Keller and collaborators investigated quasi-random sampling patterns for a variety of applications in graphics (Keller 1996, 1997, 2001). The (0, 2)-sequence sampling techniques used in the `LDSampler` and `BestCandidateSampler` are based on a paper by Kollig and Keller (2002). They are one instance of a general type of low-discrepancy sequence known as (t, s)-sequences and (t, m, s)-nets. These are discussed further by Niederreiter (1992).

BestCandidateSampler 381
LDSampler 373
Sampler 340

Some of Kollig and Keller's techniques are based on algorithms developed by Friedel and Keller (2000). Wong, Luk, and Heng (1997) compared the numeric error of various low-discrepancy sampling schemes. Keller (2001, 2006) argued that because low-discrepancy patterns tend to converge more quickly than others, they are the most efficient sampling approach for generating high-quality imagery.

Chiu, Shirley, and Wang (1994) suggested a *multijittered* 2D sampling technique that combined the properties of stratified and Latin hypercube approaches, although their technique doesn't ensure good distributions across all elementary intervals as (0, 2)-sequences do.

Mitchell (1996b) investigated how much better stratified sampling patterns are than random patterns in practice. In general, the smoother the function being sampled, the more effective they are. For very quickly changing functions (e.g., pixel regions overlapped by complex geometry), sophisticated stratified patterns perform no better than unstratified random patterns. Therefore, for scenes with complex variation in the high-dimensional image function, the advantages of fancy sampling schemes compared to a simple stratified pattern are likely to be minimal.

A unique approach to image sampling was developed by Bolin and Meyer (1995), who implemented a ray tracer that directly synthesized images in the frequency domain. This made it possible to implement interesting adaptive sampling approaches based on perceptual metrics related to the image's frequency content.

Cook (1986) first introduced the Gaussian filter to graphics. Mitchell and Netravali (1988) investigated a family of filters using experiments with human observers to find the most effective ones; the `MitchellFilter` in this chapter is the one they chose as the best. Kajiya and Ullner (1981) investigated image filtering methods that account for the effect of the reconstruction characteristics of Gaussian falloff from pixels in CRTs, and, more recently, Betrisey et al. (2000) described Microsoft's ClearType technology for display of text on LCDs.

There has been quite a bit of research into reconstruction filters for image resampling applications. Although this application is not the same as reconstructing nonuniform samples for image synthesis, much of this experience is applicable. Turkowski (1990b) reported that the Lanczos windowed sinc filter gives the best results of a number of filters for image resampling. Meijering et al. (1999) tested a variety of filters for image resampling by applying a series of transformations to images such that if perfect resampling had been done the final image would be the same as the original. They also found that the Lanczos window performed well (as did a few others) and that truncating the sinc without a window gave some of the worst results. Other work in this area includes papers by Möller et al. (1997) and Machiraju and Yagel (1996).

Kirk and Arvo (1991) identified a subtle problem with adaptive sampling algorithms: in short, if a set of samples is both used to decide if more samples should be taken and is also added to the image, the end result is *biased* and doesn't converge to the correct result in the limit. For this reason, the `Sampler::ReportResults()` method is able to indicate if a set of samples should be discarded and not reported to the `Film`. Mitchell (1987) observed that standard image reconstruction techniques fail in the presence of adaptive sampling:

Film 403
Sampler::ReportResults() 341

the contribution of a dense clump of samples in part of the filter's extent may incorrectly have a large effect on the final value purely due to the number of samples taken in that region. He described a multi-stage box filter that addresses this issue.

Since initial work in adaptive sampling by Lee et al. (1985), Kajiya (1986), and Purgathofer (1987), a number of sophisticated and effective adaptive sampling methods have been developed in recent years. Notable work includes Hachisuka et al. (2008a), who adaptively sampled in the 5D domain of image location, time, and lens position, rather than just in image location, and introduced a novel multidimensional filtering method, Shinya (1993) and Egan et al. (2009), who developed adaptive sampling and reconstruction methods focused on rendering motion blur, and Overbeck et al. (2009), who developed adaptive sampling algorithms based on wavelets for image reconstruction.

Lee and Redner (1990) first suggested using a median filter, where the median of a set of samples is used to find each pixel's value, as a noise reduction technique. Rushmeier and Ward (1994) suggested a nonlinear filter function that works well for reducing the visual impact of noise in images generated with Monte Carlo light transport algorithms. Their observation was that if a single sample is substantially brighter than all of the samples around it, then it is likely that the other nearby samples should also have detected the bright feature that caused the spike of brightness. More recently, Suykens and Willems (2000a) and Xu and Pattanaik (2005) have developed filtering techniques to reduce noise in images rendered using Monte Carlo algorithms.

Jensen and Christensen (1995) observed that it can be more effective to separate out the contributions to pixel values based on the type of illumination they represent; low-frequency indirect illumination can be filtered differently than high-frequency direct illumination, thus reducing noise in the final image. They developed an effective filtering technique based on this observation. An improvement to this approach was developed by Keller and collaborators with the *discontinuity buffer* (Keller 1998; Wald et al. 2002). In addition to filtering slowly varying quantities like indirect illumination separately from more quickly varying quantities like surface reflectance, the discontinuity buffer uses geometric quantities like the surface normal at nearby pixels to determine whether their corresponding values can be reasonably included at the current pixel. Kontkanen et al. (2004) built on these approaches to build a filtering approach for indirect illumination when using the irradiance caching algorithm (Section 15.5).

A number of different approaches have been developed for mapping out-of-gamut colors to the displayable range; see Rougeron and Péroche's survey article for discussion of this issue and references to various approaches (Rougeron and Péroche 1998). This topic was also covered by Hall (1989).

While most image file formats store 8 bits for each red, green, and blue channel (and possibly alpha), a number of new formats are also able to store floating-point data. Ward (1991b) suggested an efficient technique for compactly storing floating-point image data from realistic image synthesis and also developed an extension to the TIFF image file format for accurate and compact, high dynamic range color representation (Larson 1998). The OpenEXR format was developed by Kainz and Bogart at Industrial Light and Magic; one of its innovations is a 16-bit floating-point format (Kainz, Bogart, and Hess 2002). Peercy et al. (2000) were the first to use such a format for image representation.

The images that a renderer generates don't necessarily need to store color values at each pixel. For example, Séquin and Smyrl (1989) described a system that stored the tree of all rays traced at each pixel; this made it possible to modify light and material colors in the scene and generate a new image of the scene without needing to retrace any of the rays. Both Perlin (1985a) and Saito and Takahashi (1990) have shown a number of uses for images that stored 3D position, surface normal, and other geometric and material properties of the first visible object in each pixel, including fast reshading of the scene and illustration-inspired rendering styles.

Tone reproduction—algorithms for displaying high dynamic range images on low dynamic range display devices—became an active area of research starting with the work of Tumblin and Rushmeier (1993). Chiu et al. (1993) and Ward (1994a) soon followed with new approaches. The survey article of Devlin et al. (2002) summarizes most of the work in this area through 2002, giving pointers to the original papers. See also Reinhard et al.'s book on high dynamic range imaging, which includes significant coverage of this topic (Reinhard et al. 2005). For background information on properties of the human visual system, Wandell's book on vision is an excellent starting point (Wandell 1995). Ferwerda (2001) presented an overview of the human visual system for applications in graphics, and Malacara (2002) gave a concise overview of color theory and basic properties of how the human visual system processes color.

An interesting application of knowledge of the characteristics of the human visual system is perceptually driven rendering, where less work is done in areas of the image that are known to be visually unimportant and more work is done in areas that are known to be important. Both Bolin and Meyer (1998) and Ramasubramanian, Pattanaik, and Greenberg (1999) wrote renderers that performed adaptive sampling based on perceptual models. Walter, Pattanaik, and Greenberg (2002) developed a system that detected and took advantage of masking of noise artifacts due to textures to sample at lower rates in areas of the image where doing so wouldn't be noticeable.

EXERCISES

3 7.1 A substantially different software design for sample generation for integration is described in Keller's technical report (Keller 2001). What are the advantages and disadvantages of that approach compared to the one in `pbrt`? Modify `pbrt` to support sample generation along the lines of the approach described there and compare flexibility, performance, and image quality to the current implementation.

2 7.2 Although the `ImageFilm` always allocates storage for both `Pixel::Lxyz` and `Pixel::splatXYZ` values, the `SamplerRenderer` doesn't ever call `Film::Splat()` and thus the storage for splats is unused but is still loaded into the processor cache each time a `Pixel` is accessed. Modify the `ImageFilm` representation to separately allocate separate storage for the `splatXYZ` data, thus reducing the storage needed for pixels in the nonsplatting case to 4 `floats`, or 16 bytes. (You may still want to pad out the structure for `splatXYZ` values to 16 bytes, so that they never straddle cache lines.)

`Film::Splat()` 403
`ImageFilm` 404
`Pixel` 406
`Pixel::Lxyz` 406
`Pixel::splatXYZ` 406
`SamplerRenderer` 25

Although this separation won't save memory, it should marginally improve performance: twice as many `Pixels` can fit into a cache line, thus reducing cache misses. Measure the change in performance from this change. What is the difference for a scene with no geometry or lights (where the overhead of film update is a relatively larger part of overall runtime)? What about a more complex scene?

2 **7.3** As mentioned in Section 7.8.2, the `ImageFilm::Splat()` method doesn't use the current pixel filter but instead just splats the sample to the single pixel it's closest to, effectively using a box filter. In order to apply an arbitrary filter, the filter must be normalized so that it integrates to one over its domain; this constraint isn't currently required by `pbrt`. Modify the computation of `filterTable` in the `ImageFilm` constructor so that the tabularized function is normalized. (Don't forget that the table only stores one quarter of the function's extent when computing the normalization factor.) Then, modify the implementation of the `Splat()` method to use the current filter. Investigate the time and image-quality differences that result.

2 **7.4** An alternative design for parallelizing `pbrt` would be to have each rendering task have its own private memory for image storage. The tasks could could directly update this memory without using atomic instructions or a mutex. (Note that each task would only need storage in a small region of the image, just large enough to cover the pixel extent it is generating samples for as well as additional pixels at the boundary due to the filter function's extent.) As tasks finish, they could then acquire a mutex and merge their results in with the global image. Implement this approach in `pbrt` and determine if it noticeably affects system performance. You may want to use relatively simple scenes for testing, such that the portion of runtime spent in sample filtering and image update is relatively large with respect to overall runtime.

One additional advantage of this approach is that the tasks' images can potentially be merged in order, leading to more consistent results across multiple runs of the system. With `pbrt`'s current implementation, the pixel values in an image may change by small amounts with rerendering, due to tasks computing samples in different orders with respect to each other. A pixel that had a final value that came from samples from three different tasks, $v_1 + v_2 + v_3$, may sometimes be computed as $(v_1 + v_2) + v_3$ and sometimes as $(v_3 + v_1) + v_2$, for example. Due to floating-point round-off, these two values will in general be different. While these differences aren't normally a problem, they wreak havoc with automated testing scripts that might want to verify that a believed-to-be-innocuous change to the system didn't cause any differences in rendered images.

1 **7.5** Modify `pbrt` to create images where each pixel encodes the amount of time spent computing the sample values inside that pixel's filter extent. (A one-pixel-wide box filter is probably the most useful filter for this exercise.) Render images of a variety of scenes with this technique. What insight about the system's performance do the resulting images bring?

ImageFilm 404
ImageFilm::Splat() 411
Pixel 406

● 7.6 One of the advantages of the linearity assumption in radiometry is that the final image of a scene is the same as the sum of individual images that account for each light source's contribution (assuming a floating-point image file format is used that doesn't clip pixel radiance values). An implication of this property is that if a renderer creates a separate image for each light source, it is possible to write interactive lighting design tools that make it possible to quickly see the effects of scaling the contributions of individual lights in the scene without needing to rerender it from scratch. Instead, a light's individual image can be scaled and the final image regenerated by summing all of the light images again. (This technique was first applied for opera lighting design by Dorsey, Sillion, and Greenberg (1991).) Modify `pbrt` to output a separate image for each of the lights in the scene and write an interactive lighting design tool that uses them in this manner.

● 7.7 Mitchell and Netravali (1988) noted that there is a family of reconstruction filters that use both the value of a function and its derivative at the point to do substantially better reconstruction than if just the value of the function is known. Furthermore, they report that they have derived closed-form expressions for the screen space derivatives of Lambertian and Phong reflection models, although they do not include these expressions in their paper. Investigate derivative-based reconstruction and extend `pbrt` to support this technique. Because it will likely be difficult to derive expressions for the screen space derivatives for general shapes and BSDF models, investigate approximations based on finite differencing. Techniques built on the ideas behind the ray differentials of Section 10.1 may be fruitful for this effort.

● 7.8 Image-based rendering is the general name for a set of techniques that use one or more images of a scene to synthesize new images from viewpoints different than the original ones. One such approach is lightfield rendering, where a set of images from a densely spaced set of positions is used (Levoy and Hanrahan 1996; Gortler et al. 1996). Read these two papers on lightfields and modify `pbrt` to directly generate lightfields of scenes, without requiring that the renderer be run multiple times, once for each camera position. It will probably be necessary to write a specialized `Camera`, `Sampler`, and `Film` to do this. Also, write an interactive lightfield viewer that loads lightfields generated by your implementation and generates new views of the scene.

● 7.9 Rather than just storing spectral values in an image, it's often useful to store additional information about the objects in the scene that were visible at each pixel. See, for example, the SIGGRAPH papers by Perlin (1985a) and Saito and Takahashi (1990). For example, if the 3D position, surface normal, and BRDF of the object at each pixel are stored, then the scene can be efficiently rerendered after moving the light sources (Gershbein and Hanrahan 2000). Alternatively, if each sample stores information about all of the objects visible along its camera ray, rather than just the first one, new images from shifted viewpoints can be rerendered (Shade et al. (1998)). Investigate representations for deep frame

buffers and algorithms that use them; extend `pbrt` to support the creation of images like these and develop tools that operate on them.

❷ 7.10 Implement a median filter for image reconstruction: for each pixel, store the median of all of the samples within a filter extent around it. This task is complicated by the fact that filters in `ImageFilm` must be *linear*—the value of the filter function is determined solely by the position of the sample with respect to the pixel position, and the value of the sample has no impact on the value of the filter function. Because the implementation assumes that filters are linear, and because it doesn't store sample values after adding their contribution to the image, implementing the median filter will require generalizing the `ImageFilm` or developing a new `Film` implementation.

Render images using integrators like the `PathIntegrator` that have objectionable image noise with regular image filters. How successful is the median filter at reducing noise? Are there visual shortcomings to using the median filter? Can you implement this approach without needing to store all of the image sample values before computing final pixel values?

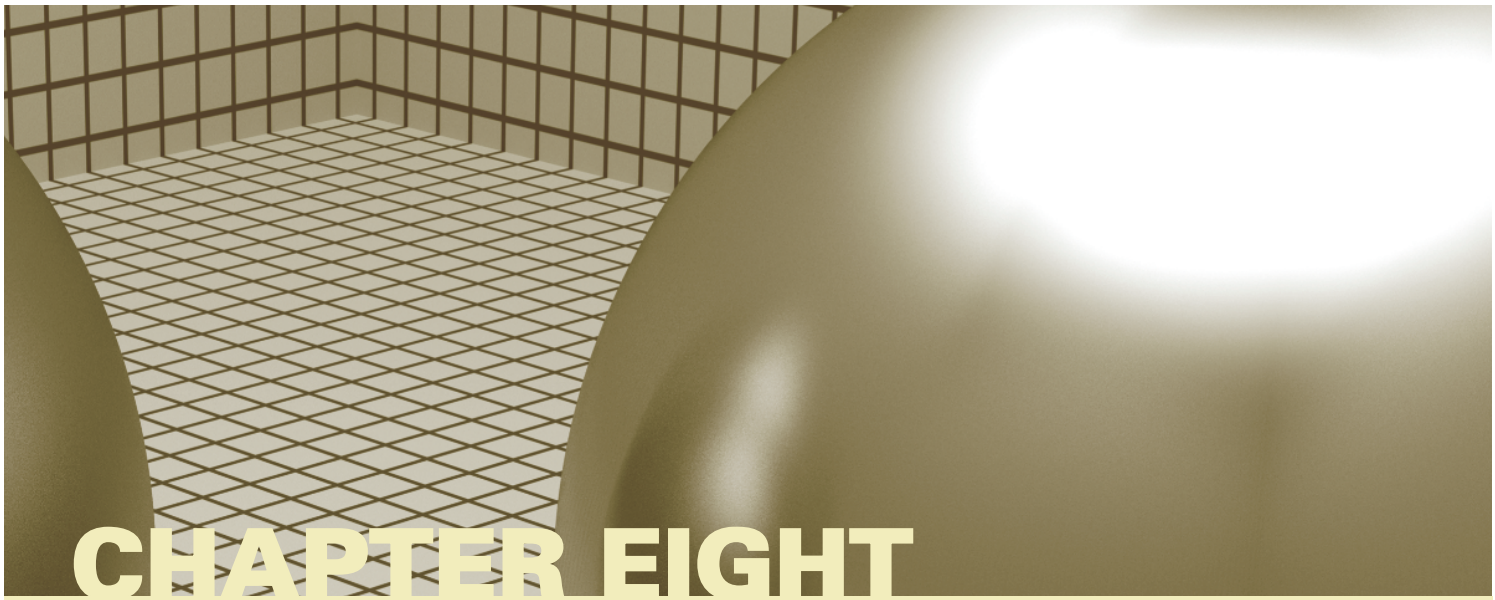
❷ 7.11 An alternative to the median filter is to discard the sample with the lowest contribution and the sample with the largest contribution in a pixel's filter region. This approach uses more of the information gathered during sampling. Implement this approach and compare the results to the median filter.

❸ 7.12 Implement the discontinuity buffer, as described by Keller and collaborators (Keller 1998; Wald et al. 2002). You will probably need to modify the `Film` interface so that geometric information about the intersection point is passed to the `Film::AddSample()` method and modify the interface to the `SurfaceIntegrators` so that they can separately return direct and indirect illumination contributions. Render images showing its effectiveness when rendering images with indirect illumination.

❸ 7.13 Implement one of the recent adaptive sampling and reconstruction techniques such as the ones described by Hachisuka et al. (2008), Egan et al. (2009), or Overbeck et al. (2009). How much more efficiently do they generate images of equal quality than just uniformly sampling at a high rate? How much more effective are they than the `AdaptiveSampler`? How do they affect running time for simple scenes where adaptive sampling isn't needed?

AdaptiveSampler 386
Film 403
Film::AddSample() 403
ImageFilm 404
PathIntegrator 766
SurfaceIntegrator 740

This page intentionally left blank



CHAPTER EIGHT



REFLECTION MODELS

This chapter defines a set of classes for describing the way that light scatters at surfaces. Recall that in Section 5.6.1 we introduced the bidirectional reflectance distribution function (BRDF) abstraction to describe light reflection at a surface, the BTDF to describe transmission at a surface, and the BSDF to encompass both of these effects. In this chapter, we will start by defining a generic interface to these surface reflection and transmission functions. Scattering from realistic surfaces is often best described as a mixture of multiple BRDFs and BTDFs; in Chapter 9, we will introduce a BSDF object that combines multiple BRDFs and BTDFs to represent overall scattering from the surface. The current chapter sidesteps the issue of reflection and transmission properties that vary over the surface; the texture classes of Chapter 10 will address that problem. Classes that represent subsurface scattering properties of objects will be introduced in Section 11.6, after some of the related theory is introduced in Chapter 11.

Surface reflection models come from a number of sources:

1. *Measured data:* Reflection distribution properties of many real-world surfaces have been measured in laboratories. This data may be used directly in tabular form or to compute coefficients for a set of basis functions.
2. *Phenomenological models:* Equations that attempt to describe the qualitative properties of real-world surfaces can be remarkably effective at mimicking them. These types of BSDFs can be particularly easy to use, since they tend to have intuitive parameters that modify their behavior (e.g., "roughness"). Many of the reflection functions used in computer graphics fall into this category.
3. *Simulation:* Sometimes, low-level information is known about the composition of a surface. For example, we might know that a paint is comprised of colored particles of some average size suspended in a medium, or that a particular fabric is comprised of two types of thread, each with known reflectance properties. In these

cases, light scattering from the microgeometry can be simulated to generate reflection data. This simulation can be done either during rendering or as a preprocess, after which it may be fit to a set of basis functions for use during rendering.

4. *Physical (wave) optics:* Some reflection models have been derived using a detailed model of light, treating it as a wave and computing the solution to Maxwell's equations to find how it scatters from a surface with known properties. These models tend to be computationally expensive, however, and usually aren't appreciably more accurate than models based on geometric optics.
5. *Geometric optics:* As with simulation approaches, if the surface's low-level scattering and geometric properties are known, then closed-form reflection models can sometimes be derived directly from these descriptions. Geometric optics makes modeling light's interaction with the surface more tractable, since complex wave effects like polarization are ignored.

In this chapter, we will describe implementations of reflection models based on measured data, phenomenological models, and geometric optics. The "Further Reading" section at the end of this chapter gives pointers to a variety of other models.

Before we define the relevant interfaces, a brief review of how they fit into the overall system is in order. In the common case where the `SamplerRenderer` is used, the renderer is first responsible for determining which surface is first visible along a ray. It then calls the integrator classes, defined in Chapter 15, which call the surface shader that is bound to the surface. The surface shader is a method in subclasses of the `Material` class and is responsible for deciding what the BSDF is at a particular point on the surface (see Chapter 9); it returns a `BSDF` object that holds BRDFs and BTDFs that it has allocated and initialized for that point. The integrator then uses the BSDF to compute the scattered light at the point, based on the incoming illumination from the light sources in the scene. (The process where the `MetropolisRenderer` is used rather than the `SamplerRenderer` is similar, but without the use of separate `Integrators`.)

Basic Terminology

In order to be able to compare the visual appearance of different reflection models, we will introduce some basic terminology for describing reflection from surfaces.

Reflection from surfaces can be split into four broad categories: *diffuse*, *glossy specular*, *perfect specular*, and *retro-reflective* (Figure 8.1). Most real surfaces exhibit reflection that is a mixture of these four types. Diffuse surfaces scatter light equally in all directions. Although a perfectly diffuse surface isn't physically realizable, examples of near-diffuse surfaces include dull chalkboards and matte paint. Glossy specular surfaces such as plastic or high-gloss paint scatter light preferentially in a set of reflected directions—they show blurry reflections of other objects. Perfect specular surfaces scatter incident light in a single outgoing direction. Mirrors and glass are examples of perfect specular surfaces. Finally, retro-reflective surfaces like velvet or the Earth's moon scatter light primarily back along the incident direction. Images throughout this chapter will show the differences between these various types of reflection when used in rendered scenes.

BSDF 478
Integrator 740
Material 483
MetropolisRenderer 852
SamplerRenderer 25

Given a particular category of reflection, the reflectance distribution function may be *isotropic* or *anisotropic*. Most objects are isotropic: if you choose a point on the surface and rotate it around its normal axis at that point, the amount of light reflected doesn't

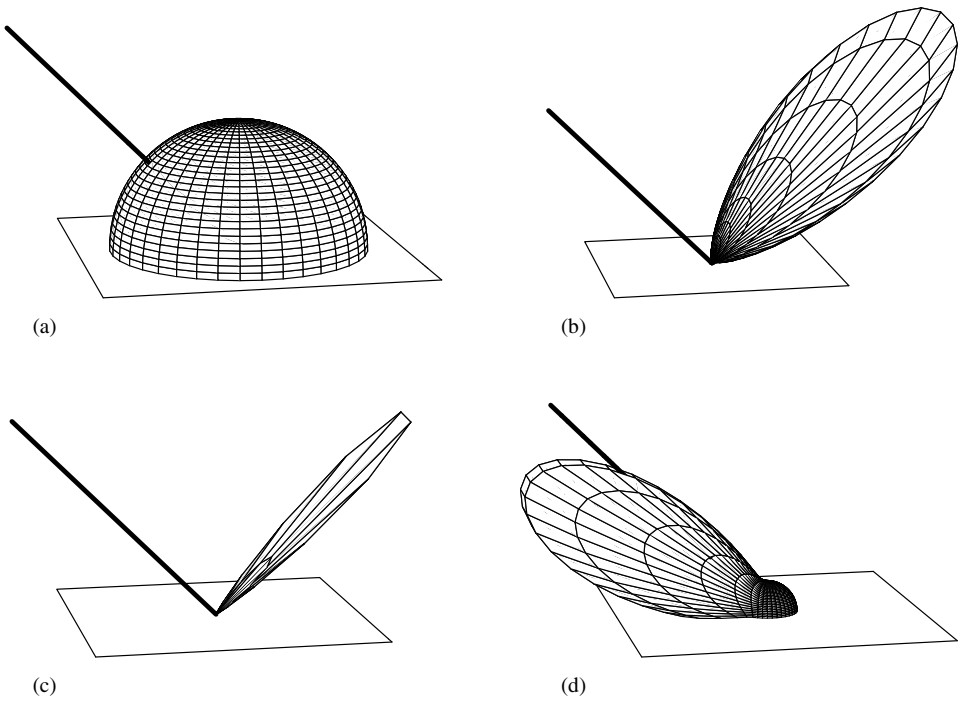


Figure 8.1: Reflection from a surface can be generally categorized by the distribution of reflected light from an incident direction (heavy lines): (a) diffuse, (b) glossy specular, (c) perfect specular, and (d) retro-reflective distributions.

change. In contrast, anisotropic materials reflect different amounts of light as you rotate them in this way. Examples of anisotropic surfaces include brushed metal, phonographic records, and compact disks.

Geometric Setting

Reflection computations in `pbrt` are evaluated in a reflection coordinate system where the two tangent vectors and the normal vector at the point being shaded are aligned with the x, y, and z axes, respectively (Figure 8.2). All direction vectors passed to and returned from the BRDF and BTDF routines will be defined with respect to this coordinate system. It is important to understand this coordinate system in order to understand the BRDF and BTDF implementations in this chapter.

The shading coordinate system also gives a frame for expressing directions in spherical coordinates (θ, ϕ); the angle θ is measured from the given direction to the z axis, and ϕ is the angle formed with the x axis after projection of the direction onto the xy plane. Given a direction vector ω in this coordinate system, it is easy to compute quantities like the cosine of the angle that it forms with the normal direction:

$$\cos\theta = (\mathbf{n} \cdot \omega) = ((0, 0, 1) \cdot \omega) = \omega_z.$$

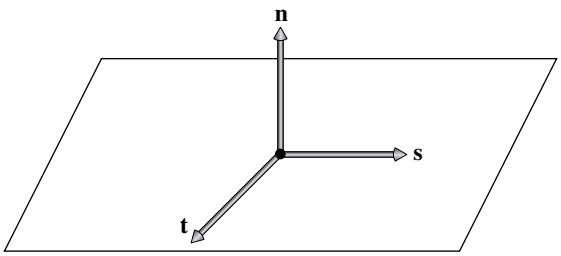


Figure 8.2: The Basic BSDF Interface Setting. The shading coordinate system is defined by the orthonormal basis vectors $(\mathbf{s}, \mathbf{t}, \mathbf{n})$. We will orient these vectors such that they lie along the x, y, and z axes in this coordinate system. Direction vectors ω in world space are transformed into the shading coordinate system before any of the BRDF or BTDF methods are called.

We will provide utility functions to compute this value and its absolute value; their use serves to make the expression of BRDF and BTDF implementations more clear.

⟨*BSDF Inline Functions*⟩ ≡
```
inline float CosTheta(const Vector &w) { return w.z; }
inline float AbsCosTheta(const Vector &w) { return fabsf(w.z); }
```

The value of $\sin^2\theta$ can be computed using the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$.

⟨*BSDF Inline Functions*⟩ +≡
```
inline float SinTheta2(const Vector &w) {
    return max(0.f, 1.f - CosTheta(w)*CosTheta(w));
}
```

⟨*BSDF Inline Functions*⟩ +≡
```
inline float SinTheta(const Vector &w) {
    return sqrtf(SinTheta2(w));
}
```

We can similarly use the shading coordinate system to simplify the calculations for the sine and cosine of the ϕ angle (Figure 8.3). In the plane of the point being shaded, the vector ω has coordinates (x, y), which are given by $r\cos\phi$ and $r\sin\phi$, respectively. The radius r is $\sin\theta$, so

$$\cos\phi = \frac{x}{r} = \frac{x}{\sin\theta}$$
$$\sin\phi = \frac{y}{r} = \frac{y}{\sin\theta}.$$

CosTheta() 426
Vector 57

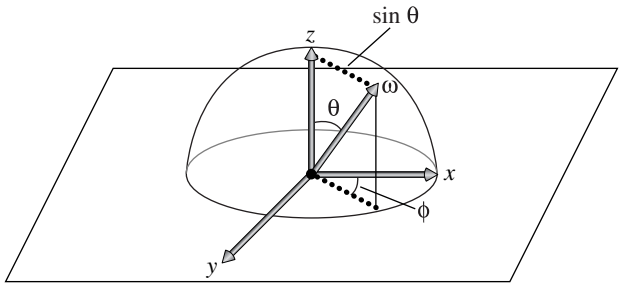


Figure 8.3: The value of $\sin\theta$ can be found by computing the length of the dotted line, which is the magnitude of the xy components of the vector. (Recall that the length of the vector ω is one.) The $\sin\phi$ and $\cos\phi$ values can be computed using the circular coordinate equations $x = r\cos\phi$ and $y = r\sin\phi$, where r, the length of the dashed line, is equal to $\sin\theta$.

⟨*BSDF Inline Functions*⟩ +≡

```
inline float CosPhi(const Vector &w) {
    float sintheta = SinTheta(w);
    if (sintheta == 0.f) return 1.f;
    return Clamp(w.x / sintheta, -1.f, 1.f);
}
inline float SinPhi(const Vector &w) {
    float sintheta = SinTheta(w);
    if (sintheta == 0.f) return 0.f;
    return Clamp(w.y / sintheta, -1.f, 1.f);
}
```

Another convention we will follow is that the incident light direction ω_i and the outgoing viewing direction ω_o will both be normalized and outward facing after being transformed into the local coordinate system at the surface. By convention, the surface normal $\mathbf{n}$ always points to the "outside" of the object, which makes it easy to determine if light is entering or exiting transmissive objects: if the incident light direction ω_i is in the same hemisphere as $\mathbf{n}$, then light is entering; otherwise, it is exiting.

Therefore, one detail to keep in mind is that the normal may be on the opposite side of the surface than one or both of the ω_i and ω_o direction vectors. Unlike many other renderers, `pbrt` does not flip the normal to lie on the same side as ω_o. Therefore, it is important that BRDFs and BTDFs be implemented so that they don't expect otherwise.

Furthermore, note that the local coordinate system used for shading may not be exactly the same as the coordinate system returned by the `Shape::Intersect()` routines from Chapter 3; they can be modified between intersection and shading to achieve effects like bump mapping. See Chapter 9 for examples of this kind of modification.

Clamp() 1000
Shape::Intersect() 111
SinTheta() 426
Vector 57

One final detail to be aware of when reading this chapter is that BRDF and BTDF implementations should not concern themselves with whether ω_i and ω_o lie in the same hemisphere. For example, although a reflective BRDF should in principle detect if the incident direction is above the surface and the outgoing direction is below and always return no reflection in this case, here we will expect the reflection function to instead compute and return the amount of light reflected using the appropriate formulas for their reflection model, ignoring the detail that they are not in the same hemisphere. Higher-level code in `pbrt` will ensure that only reflective or transmissive scattering routines are evaluated as appropriate. The value of this convention will be explained in Section 9.1.

8.1 BASIC INTERFACE

We will first define the interface for the individual BRDF and BTDF functions. BRDFs and BTDFs share a common base class, `BxDF`. Because both have the exact same interface, sharing the same base class reduces repeated code and allows some parts of the system to work with `BxDF`s generically without distinguishing between BRDFs and BTDFs.

⟨*BxDF Declarations*⟩ ≡
```
class BxDF {
public:
    ⟨BxDF Interface 429⟩
    ⟨BxDF Public Data 429⟩
};
```

The `BSDF` class, which will be introduced in Section 9.1, holds a collection of `BxDF` objects that together describe the scattering at a point on a surface. Although we are hiding the implementation details of the `BxDF` behind a common interface for reflective and transmissive materials, some of the light transport algorithms in Chapter 15 will need to distinguish between these two types. Therefore, all `BxDF`s have a `BxDF::type` member that holds flags from `BxDFType`. For each `BxDF`, the flags should have exactly one of `BSDF_REFLECTION` or `BSDF_TRANSMISSION` set, and exactly one of the diffuse, glossy, and specular flags. Note that there is no retro-reflective flag; retro-reflection is treated as glossy reflection in this categorization.

⟨*BSDF Declarations*⟩ ≡
```
enum BxDFType {
    BSDF_REFLECTION   = 1<<0,
    BSDF_TRANSMISSION = 1<<1,
    BSDF_DIFFUSE      = 1<<2,
    BSDF_GLOSSY       = 1<<3,
    BSDF_SPECULAR     = 1<<4,
    BSDF_ALL_TYPES        = BSDF_DIFFUSE |
                            BSDF_GLOSSY |
                            BSDF_SPECULAR,
    BSDF_ALL_REFLECTION   = BSDF_REFLECTION |
                            BSDF_ALL_TYPES,
    BSDF_ALL_TRANSMISSION = BSDF_TRANSMISSION |
                            BSDF_ALL_TYPES,
```

BSDF 478
BSDF_ALL_REFLECTION 428
BSDF_ALL_TRANSMISSION 428
BSDF_ALL_TYPES 428
BSDF_DIFFUSE 428
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDF 428
BxDF::type 429
BxDFType 428

```
    BSDF_ALL              = BSDF_ALL_REFLECTION |
                            BSDF_ALL_TRANSMISSION
};
```

⟨*BxDF Public Data*⟩ ≡ 428
```
const BxDFType type;
```

⟨*BxDF Interface*⟩ ≡ 428
```
BxDF(BxDFType t) : type(t) { }
```

The `MatchesFlags()` utility method determines if the `BxDF` matches the user-supplied flags:

⟨*BxDF Interface*⟩ +≡ 428
```
bool MatchesFlags(BxDFType flags) const {
    return (type & flags) == type;
}
```

The key method that `BxDF`s provide is the `BxDF::f()` method. It returns the value of the distribution function for the given pair of directions. This interface implicitly assumes that light in different wavelengths is *decoupled*—energy at one wavelength will not be reflected at a different wavelength. By making this assumption, the effect of the reflection function can be represented directly with a `Spectrum`. To support fluorescent materials where this assumption is not true would require that this method return an $n \times n$ matrix that encoded the transfer of energy between spectral samples (where n is the number of samples in the `Spectrum` representation).

⟨*BxDF Interface*⟩ +≡ 428
```
virtual Spectrum f(const Vector &wo, const Vector &wi) const = 0;
```

Not all `BxDF`s can be evaluated with the `f()` method. For example, perfectly specular objects like a mirror, glass, or water only scatter light from a single incident direction into a single outgoing direction. Such `BxDF`s are best described with delta distributions that are zero except for the single direction where light is scattered.

These `BxDF`s need special handling in `pbrt`, so we will also provide the method `BxDF::Sample_f()`. This method is used both for handling scattering that is described by delta distributions as well as for randomly sampling directions from `BxDF`s that scatter light along multiple directions; this second application will be explained in the discussion of Monte Carlo sampling in Chapter 14. `BxDF::Sample_f()` computes the direction of incident light ω_i given an outgoing direction ω_o and returns the value of the `BxDF` for the given pair of directions. For delta distributions, it is necessary for the `BxDF` to choose the incident light direction in this way, since the caller has no chance of generating the appropriate ω_i direction.[1] The `u1`, `u2`, and `pdf` parameters aren't needed for delta distribution `BxDF`s, so they will be explained later, in Section 14.5, when we provide implementations of this method for nonspecular reflection functions.

`BxDF` 428
`BxDF::f()` 429
`BxDF::Sample_f()` 694
`BxDF::type` 429
`BxDFType` 428
`Spectrum` 263
`Vector` 57

1 Delta distributions in reflection functions have some additional subtle implications for light transport algorithms. Sections 14.5.4 and 15.2.5 describe the issues in detail.

⟨*BxDF Interface*⟩ +≡ 428
```
virtual Spectrum Sample_f(const Vector &wo, Vector *wi,
                          float u1, float u2, float *pdf) const;
```

8.1.1 REFLECTANCE

It can be useful to take the aggregate behavior of the 4D BRDF or BTDF, defined as a function over pairs of directions, and reduce it to a 2D function over a single direction, or even to a constant value that describes its overall scattering behavior.

The *hemispherical-directional reflectance* is a 2D function that gives the total reflection in a given direction due to constant illumination over the hemisphere, or, equivalently, total reflection over the hemisphere due to light from a given direction.[2] It is defined as

$$\rho_{\mathrm{hd}}(\omega_{\mathrm{o}}) = \int_{\mathcal{H}^2(\mathbf{n})} f_{\mathrm{r}}(\mathrm{p}, \omega_{\mathrm{o}}, \omega_{\mathrm{i}}) \, |\cos\theta_{\mathrm{i}}| \, \mathrm{d}\omega_{\mathrm{i}}.$$

The `BxDF::rho()` method computes the reflectance function ρ_{hd}. Some `BxDF`s can compute this value in closed form, although most use Monte Carlo integration to compute an approximation to it. For those `BxDF`s, the `nSamples` and `samples` parameters are used by the implementation of the Monte Carlo algorithm; they are explained in Section 14.5.5.

⟨*BxDF Interface*⟩ +≡ 428
```
virtual Spectrum rho(const Vector &wo, int nSamples,
                     const float *samples) const;
```

The *hemispherical-hemispherical reflectance* of a surface, denoted by ρ_{hh}, is a constant spectral value that gives the fraction of incident light reflected by a surface when the incident light is the same from all directions. It is

$$\rho_{\mathrm{hh}} = \frac{1}{\pi} \int_{\mathcal{H}^2(\mathbf{n})} \int_{\mathcal{H}^2(\mathbf{n})} f_{\mathrm{r}}(\mathrm{p}, \omega_{\mathrm{o}}, \omega_{\mathrm{i}}) \, |\cos\theta_{\mathrm{o}} \cos\theta_{\mathrm{i}}| \, \mathrm{d}\omega_{\mathrm{o}} \mathrm{d}\omega_{\mathrm{i}}.$$

The `BxDF::rho()` method computes ρ_{hh} if no direction ω_{o} is provided. The remaining parameters are again used when computing a Monte Carlo estimate, if needed.

⟨*BxDF Interface*⟩ +≡ 428
```
virtual Spectrum rho(int nSamples, const float *samples1,
                     const float *samples2) const;
```

8.1.2 BRDF → BTDF ADAPTER

It's handy to define an adapter class that makes it easy to reuse an already-defined BRDF class as a BTDF, especially for phenomenological models that may be equally plausible models of transmission. The `BRDFToBTDF` class takes a BRDF's pointer in the constructor and uses it to implement a BTDF. In particular, doing so involves forwarding method calls on to the BRDF and switching the ω_{i} direction to lie in the other hemisphere.

BRDFToBTDF 431
BxDF 428
BxDF::rho() 430
Spectrum 263
Vector 57

2 The fact that these two quantities are equal is due to the reciprocity of real-world reflection functions. If we had a nonphysically based BRDF that did not obey reciprocity, this assumption, along with many others in `pbrt`, would break down.

⟨*BxDF Declarations*⟩ +≡
```
class BRDFToBTDF : public BxDF {
public:
    ⟨BRDFToBTDF Public Methods 431⟩
private:
    BxDF *brdf;
};
```

The constructor for the adapter class is simple. It switches the reflection and transmission flags of the `BxDF::type` member.

⟨*BRDFToBTDF Public Methods*⟩ ≡ 431
```
BRDFToBTDF(BxDF *b)
    : BxDF(BxDFType(b->type ^ (BSDF_REFLECTION | BSDF_TRANSMISSION))) {
    brdf = b;
}
```

The adapter needs to convert an incoming vector to the corresponding vector in the opposite hemisphere. Fortunately, this is a simple calculation in the shading coordinate system, just requiring negation of the vector's z coordinate.

⟨*BRDFToBTDF Public Methods*⟩ +≡ 431
```
static Vector otherHemisphere(const Vector &w) {
    return Vector(w.x, w.y, -w.z);
}
```

The `BRDFToBTDF::otherHemisphere()` method is used to reflect a ray into the other hemisphere before calling the BRDF's `BxDF::f()`, `BxDF::rho()`, and `BxDF::Sample_f()` methods. We'll only include `f()` here since the others are analogous.

⟨*BxDF Method Definitions*⟩ ≡
```
Spectrum BRDFToBTDF::f(const Vector &wo, const Vector &wi) const {
    return brdf->f(wo, otherHemisphere(wi));
}
```

BRDFToBTDF 431
BRDFToBTDF::brdf 431
BRDFToBTDF::otherHemisphere() 431
BSDF 478
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDF 428
BxDF::f() 429
BxDF::rho() 430
BxDF::Sample_f() 694
BxDF::type 429
BxDFType 428
MixMaterial 488
ScaledBxDF 431
Spectrum 263
Vector 57

8.1.3 BxDF SCALING ADAPTER

It is also useful to take a given `BxDF` and scale its contribution with a `Spectrum` value. The `ScaledBxDF` wrapper holds a `BxDF *` and a `Spectrum` and implements this functionality. This class is used by the `MixMaterial` (defined in Section 9.2.3), which creates `BSDF`s based on a weighted combination of two other materials.

⟨*BxDF Declarations*⟩ +≡
```
class ScaledBxDF : public BxDF {
public:
    ⟨ScaledBxDF Public Methods 432⟩
private:
    BxDF *bxdf;
    Spectrum s;
};
```

⟨*ScaledBxDF Public Methods*⟩ ≡ 431
```
ScaledBxDF(BxDF *b, const Spectrum &sc)
    : BxDF(BxDFType(b->type)), bxdf(b), s(sc) {
}
```

The implementations of the `ScaledBxDF` methods are straightforward; we'll only include `f()` here.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum ScaledBxDF::f(const Vector &wo, const Vector &wi) const {
    return s * bxdf->f(wo, wi);
}
```

8.2 SPECULAR REFLECTION AND TRANSMISSION

The behavior of light at perfectly smooth surfaces is relatively easy to characterize analytically using both the physical and geometric optics models. These surfaces exhibit perfect specular reflection and transmission of incident light; for a given ω_i direction, all light is scattered in a single outgoing direction ω_o. For specular reflection, this direction is the outgoing direction that makes the same angle with the normal as the incoming direction:

$$\theta_i = \theta_o.$$

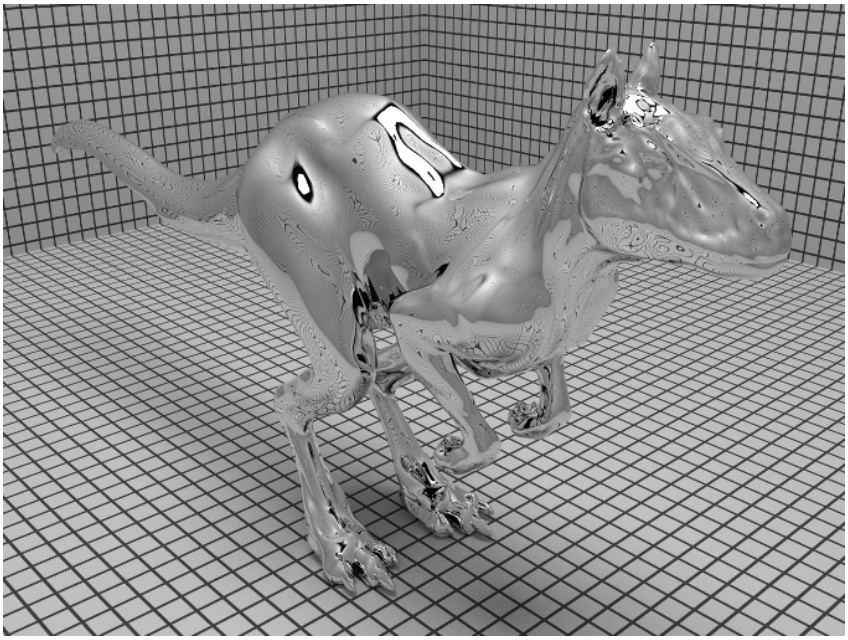
For transmission, the outgoing direction is given by *Snell's law*, which relates the angle θ_t between the transmitted direction and the surface normal $\mathbf{n}$ to the angle θ_i between the incident ray and the surface normal $\mathbf{n}$. (One of the exercises at the end of this chapter is to derive Snell's law using Fermat's principle from optics.) Snell's law is based on the *index of refraction* for the medium that the incident ray is in and the index of refraction for the medium it is entering. The index of refraction describes how much more slowly light travels in a particular medium than in a vacuum. We will use the Greek letter η, pronounced "eta," to denote the index of refraction. Snell's law is

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t.$$

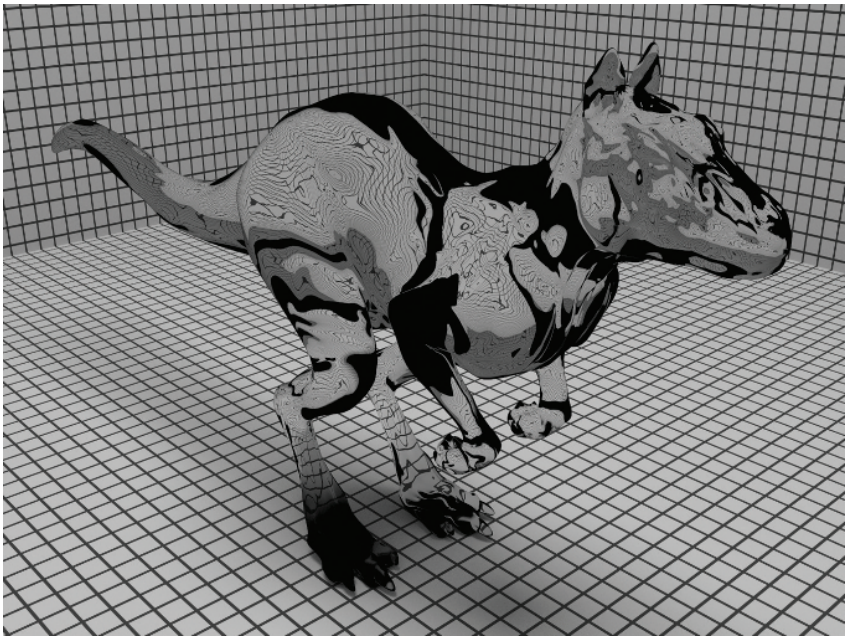
In general, the index of refraction varies with the wavelength of light. Thus, incident light generally scatters in multiple directions at the boundary between two different media, an effect known as *dispersion*. This effect can be seen when incident white light is split into spectral components by a prism. Common practice in graphics is to ignore this wavelength dependence, since this effect is generally not crucial for visual accuracy and ignoring it simplifies light transport calculations substantially. Alternatively, the paths of multiple beams of light (e.g., at a series of discrete wavelengths) can be tracked through the environment in which a dispersive object is found. The "Further Reading" section at the end of this chapter has pointers to more information on these issues.

Figure 8.4 shows the effect when the Killeroo model is rendered with a BRDF describing perfect specular reflection and a BTDF describing specular transmission. Note how refraction through the transmissive object distorts the scene behind it.

BxDF 428
BxDF::f() 429
BxDFType 428
ScaledBxDF 431
ScaledBxDF::bxdf 431
ScaledBxDF::s 431
Spectrum 263
Vector 57



(a)



(b)

Figure 8.4: Killeroo model rendered with (a) perfect specular reflection and (b) perfect specular refraction. *(Model courtesy of headus/Rezard.)*

Table 8.1: Indices of refraction for a variety of objects, giving the ratio of the speed of light in a vacuum to the speed of light in the medium. These are generally wavelength-dependent quantities; these values are averages over the visible wavelengths.

Medium	Index of refraction η
Vacuum	1.0
Air at sea level	1.00029
Ice	1.31
Water (20°C)	1.333
Fused quartz	1.46
Glass	1.5–1.6
Sapphire	1.77
Diamond	2.42

8.2.1 FRESNEL REFLECTANCE

In addition to the reflected and transmitted directions, it is also necessary to compute the fraction of incoming light that is reflected or transmitted. In simple ray tracers, these fractions are typically just given as "reflectivity" or "transmissiveness" values, which are uniform over the entire surface. For physical reflection or refraction, however, these terms are directionally dependent and cannot be captured by constant per-surface scaling amounts. The *Fresnel equations* describe the amount of light reflected from a surface; they are the solution to Maxwell's equations at smooth surfaces.

There are two sets of Fresnel equations: one for *dielectric media* (objects that don't conduct electricity, like glass) and one for *conductors* (like metals). For each of these cases, the Fresnel equations have two forms, depending on the polarization of the incident light. Properly accounting for polarization in rendering is a complex task, and so in `pbrt` we will make the common assumption that light is unpolarized; that is, it is randomly oriented with respect to the light wave. With this simplifying assumption, the Fresnel reflectance is the average of the squares of the parallel and perpendicular polarization terms.

To compute the Fresnel reflectance of a dielectric, we need to know the indices of refraction for the two media. Table 8.1 has the indices of refraction for a number of dielectric materials. A close approximation to the Fresnel reflectance formulae for dielectrics is

$$r_{\parallel} = \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}$$
$$r_{\perp} = \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t},$$

where $r_{\parallel}$ is the Fresnel reflectance for parallel polarized light and $r_{\perp}$ is the reflectance for perpendicular polarized light. η_i and η_t are the indices of refraction for the incident and transmitted media, and ω_i and ω_t are the incident and transmitted directions, where ω_t was computed with Snell's law.

The cosine terms should all be greater than or equal to zero; for the purposes of computing these values, the geometric normal should be flipped to be on the same side as ω_i and then ω_t.

For unpolarized light, the Fresnel reflectance is

$$F_r = \frac{1}{2}(r_\parallel^2 + r_\perp^2).$$

The function `FrDiel()` computes the Fresnel reflection formula for dielectric materials and circularly polarized light. The quantities $\cos\theta_i$ and $\cos\theta_t$ are passed in with the variables `cosi` and `cost`.

⟨*BxDF Utility Functions*⟩ ≡
```
Spectrum FrDiel(float cosi, float cost, const Spectrum &etai,
                const Spectrum &etat) {
    Spectrum Rparl = ((etat * cosi) - (etai * cost)) /
                     ((etat * cosi) + (etai * cost));
    Spectrum Rperp = ((etai * cosi) - (etat * cost)) /
                     ((etai * cosi) + (etat * cost));
    return (Rparl*Rparl + Rperp*Rperp) / 2.f;
}
```

Due to conservation of energy, the energy transmitted by a dielectric is $1 - F_r$.

Unlike dielectrics, conductors don't transmit light, but some of the incident light is absorbed by the material and turned into heat. The Fresnel formula for conductors tells how much is reflected—the amount depends on the additional quantities η, the index of refraction of the conductor, and k, its *absorption coefficient*. Figure 8.5 shows a plot of the index of refraction and absorption coefficent for gold; both of these are wavelength-dependent quantities. The directory `scenes/spds/metals` in the `pbrt` distribution has wavelength-dependent data for η and k for a variety of metals. Figure 9.4 in the next chapter shows a model rendered with a metal material.

A widely used approximation to the Fresnel reflectance for conductors is

$$r_\parallel^2 = \frac{(\eta^2 + k^2)\cos\theta_i^2 - 2\eta\cos\theta_i + 1}{(\eta^2 + k^2)\cos\theta_i^2 + 2\eta\cos\theta_i + 1} \quad [8.1]$$

$$r_\perp^2 = \frac{(\eta^2 + k^2) - 2\eta\cos\theta_i + \cos\theta_i^2}{(\eta^2 + k^2) + 2\eta\cos\theta_i + \cos\theta_i^2}. \quad [8.2]$$

⟨*BxDF Utility Functions*⟩ +≡
```
Spectrum FrCond(float cosi, const Spectrum &eta, const Spectrum &k) {
    Spectrum tmp = (eta*eta + k*k) * cosi*cosi;
    Spectrum Rparl2 = (tmp - (2.f * eta * cosi) + 1) /
                      (tmp + (2.f * eta * cosi) + 1);
    Spectrum tmp_f = eta*eta + k*k;
    Spectrum Rperp2 =
        (tmp_f - (2.f * eta * cosi) + cosi*cosi) /
        (tmp_f + (2.f * eta * cosi) + cosi*cosi);
    return (Rparl2 + Rperp2) / 2.f;
}
```

FrDiel() 435
Spectrum 263

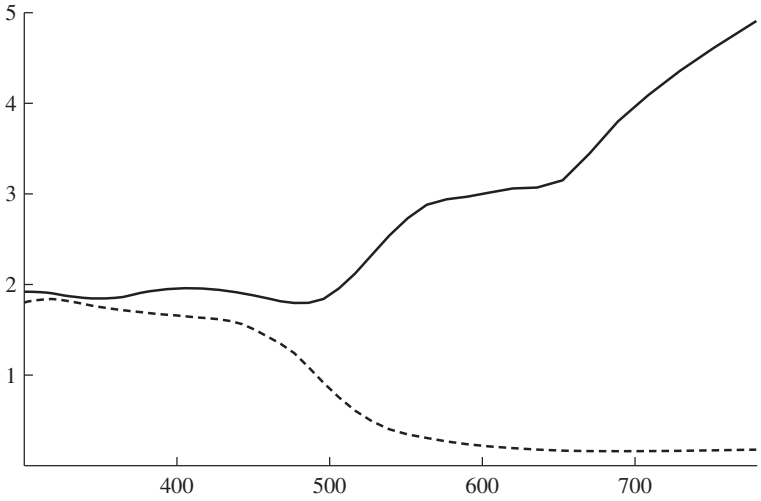


Figure 8.5: Absorption Coefficient and Index of Refraction of Gold. This plot shows the spectrally varying values of the absorption coefficient k (solid line) and the index of refraction η (dashed line) for gold, where the horizontal axis is wavelength in nm.

For convenience, we will define an abstract `Fresnel` class that provides an interface for computing Fresnel reflection coefficients. The `FresnelConductor` and `FresnelDielectric` implementations of this interface help simplify the implementation of subsequent BRDFs that may need to support both forms.

```
⟨BxDF Declarations⟩ +≡
   class Fresnel {
   public:
       ⟨Fresnel Interface 436⟩
   };
```

The only method provided by the `Fresnel` interface is `Fresnel::Evaluate()`. Given the cosine of the angle made by the incoming direction and the surface normal, it returns the amount of light reflected by the surface.

```
⟨Fresnel Interface⟩ ≡                                              436
   virtual Spectrum Evaluate(float cosi) const = 0;
```

Fresnel Conductors

`FresnelConductor` implements this interface for conductors.

Fresnel 436
Fresnel::Evaluate() 436
FresnelConductor 436
FresnelDielectric 437
Spectrum 263

```
⟨BxDF Declarations⟩ +≡
   class FresnelConductor : public Fresnel {
   public:
       ⟨FresnelConductor Public Methods 437⟩
```

```
private:
    Spectrum eta, k;
};
```

Its constructor stores the given index of refraction η and absorption coefficient k.

⟨*FresnelConductor Public Methods*⟩ ≡ 436
```
FresnelConductor(const Spectrum &e, const Spectrum &kk)
    : eta(e), k(kk) {
}
```

The evaluation routine for `FresnelConductor` is also simple; it just calls the `FrCond()` function defined earlier. Note that it takes the absolute value of `cosi` before calling `FrCond()`, since `FrCond()` expects that the cosine will be measured with respect to the normal on the same side of the surface as ω_i, or equivalently that the absolute value of $\cos\theta_i$ should be used.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum FresnelConductor::Evaluate(float cosi) const {
    return FrCond(fabsf(cosi), eta, k);
}
```

Fresnel Dielectrics

`FresnelDielectric` similarly implements the `Fresnel` interface for dielectric materials.

⟨*BxDF Declarations*⟩ +≡
```
class FresnelDielectric : public Fresnel {
public:
    ⟨FresnelDielectric Public Methods 437⟩
private:
    float eta_i, eta_t;
};
```

Its constructor stores the indices of refraction on the two sides of the surface, η_i and η_t:

⟨*FresnelDielectric Public Methods*⟩ ≡ 437
```
FresnelDielectric(float ei, float et) : eta_i(ei), eta_t(et) { }
```

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum FresnelDielectric::Evaluate(float cosi) const {
    ⟨Compute Fresnel reflectance for dielectric 438⟩
}
```

FrCond() 435
Fresnel 436
FresnelConductor 436
FresnelDielectric 437
FresnelDielectric::eta_i 437
FresnelDielectric::eta_t 437
Spectrum 263

Evaluating the Fresnel formula for dielectric media is a bit more complicated than for conductors. First, it is necessary to determine if the incident direction is on the outside of the medium or inside it, so that the two indices of refraction can be interpreted appropriately. Next, Snell's law is applied to compute the sine of the angle between the transmitted direction and the surface normal. Finally, the cosine of this angle is found using the identity $\sin^2\theta + \cos^2\theta = 1$.

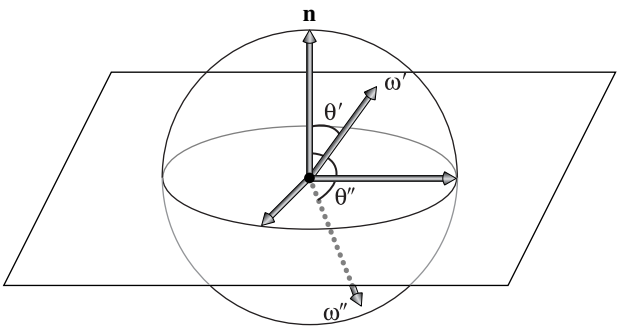


Figure 8.6: The cosine of the angle θ between a direction ω and the geometric surface normal indicates whether the direction is pointing outside the surface (in the same hemisphere as the normal) or inside the surface. In the standard reflection coordinate system, this test just requires checking the z component of the direction vector. Here, ω' is in the upper hemisphere, with a positive-valued cosine, while ω'' is in the lower hemisphere.

⟨*Compute Fresnel reflectance for dielectric*⟩ ≡ 437
```
cosi = Clamp(cosi, -1.f, 1.f);
⟨Compute indices of refraction for dielectric 438⟩
⟨Compute sint using Snell's law 438⟩
if (sint >= 1.) {
    ⟨Handle total internal reflection 439⟩
}
else {
    float cost = sqrtf(max(0.f, 1.f - sint*sint));
    return FrDiel(fabsf(cosi), cost, ei, et);
}
```

The sign of the cosine of the incident angle indicates on which side of the medium the incident ray lies (Figure 8.6). If the cosine is between 0 and 1, the ray is on the outside, and if the cosine is between −1 and 0, the ray is on the inside. The variables `ei` and `et` are set such that `ei` has the index of refraction of the incident medium.

⟨*Compute indices of refraction for dielectric*⟩ ≡ 438
```
bool entering = cosi > 0.;
float ei = eta_i, et = eta_t;
if (!entering)
    swap(ei, et);
```

Once the indices of refraction are set, we can compute $\sin\theta_t$ using Snell's law:

⟨*Compute* `sint` *using Snell's law*⟩ ≡ 438
```
float sint = ei/et * sqrtf(max(0.f, 1.f - cosi*cosi));
```

When light is traveling from one medium to another medium with a lower index of refraction, none of the light at incident angles near grazing passes into the other medium.

Clamp() 1000
FrDiel() 435

The largest angle at which this happens is called the *critical angle*; when θ_i is greater than the critical angle, *total internal reflection* occurs, and all of the light is reflected. That case is detected here by a value of $\sin\theta_t$ greater than one; in that case, the Fresnel equations are unnecessary.

⟨*Handle total internal reflection*⟩ ≡ 438
```
return 1.;
```

A Special Fresnel Interface

The `FresnelNoOp` implementation of the `Fresnel` interface returns 100% reflection for all incoming directions. Although this is physically implausible, it is a convenient capability to have available.

⟨*BxDF Declarations*⟩ +≡
```
class FresnelNoOp : public Fresnel {
public:
    Spectrum Evaluate(float) const { return Spectrum(1.); }
};
```

8.2.2 SPECULAR REFLECTION

We can now implement the `SpecularReflection` class, which describes physically plausible specular reflection, using the Fresnel interface to compute the fraction of light that is reflected. First, we will derive the BRDF that describes specular reflection. Since the Fresnel equations give the fraction of light reflected, $F_r(\omega_i)$, then we need a BRDF such that

$$L_o(\omega_o) = f_r(\omega_o, \omega_i) L_i(\omega_i) = F_r(\omega_i)\, L_i(\omega_i),$$

where ω_i is the reflection vector for ω_o about the surface normal. (Recall that $\theta_i = \theta_o$ for specular reflection, and therefore $F_r(\omega_o) = F_r(\omega_i)$.)

Such a BRDF can be constructed using the Dirac delta distribution. Recall from Section 7.1 that the delta distribution has the useful property that

$$\int f(x)\, \delta(x - x_0)\, \mathrm{d}x = f(x_0). \qquad [8.3]$$

The delta distribution requires special handling compared to standard functions. In particular, integrals with delta distributions must be evaluated by explicitly accounting for the delta distribution; their values cannot be properly computed without doing so. For example, consider the delta distribution in Equation (8.3): if we tried to evaluate it using the trapezoid rule or some other numerical integration technique, by definition of the delta distribution there would be zero probability that any of the evaluation points x_i would have a nonzero value of $\delta(x_i)$. Rather, we must allow the delta distribution to determine the evaluation point itself. We will see this issue in practice for specular `BxDF`s as well as for some of the light sources in Chapter 12.

Intuitively, we want the BRDF to be zero everywhere except at the perfect reflection direction, which suggests the use of the delta distribution. A first guess might be to use

BxDF 428
Fresnel 436
FresnelNoOp 439
Spectrum 263
SpecularReflection 440

delta functions to restrict the incident direction to the reflection angle ω_r. This would yield a BRDF of

$$f_r(p, \omega_o, \omega_i) = \delta(\omega_i - \omega_r) = \delta(\cos\theta_i - \cos\theta_r)\delta(\phi_i - \phi_r).$$

Although this seems appealing, plugging into the scattering equation, Equation (5.8), reveals a problem:

$$\begin{aligned} L_o\left(\theta_o, \phi_o\right) &= \int_{S^2} \delta(\cos\theta_i - \cos\theta_r)\delta(\phi_o - \phi_r \pm \pi) L_i\left(\theta_i, \phi_i\right) |\cos\theta_i| \, d\omega_i \\ &= L_i\left(\theta_r, \phi_r \pm \pi\right) |\cos\theta_i|. \end{aligned}$$

This is not correct because it contains an extra factor of $\cos\theta_i$. But we can divide out this factor to find the correct BRDF for perfect specular reflection:

$$f_r(p, \omega_o, \omega_i) = F_r(\omega_o) \frac{\delta(\omega_i - R(\omega_o, \mathbf{n}))}{|\cos\theta_i|},$$

where $R(\omega_o, \mathbf{n})$ is the specular reflection vector for ω_o reflected about the surface normal $\mathbf{n}$.

⟨*BxDF Declarations*⟩ +≡
```
class SpecularReflection : public BxDF {
public:
    ⟨SpecularReflection Public Methods 440⟩
private:
    ⟨SpecularReflection Private Data 440⟩
};
```

The `SpecularReflection` class takes a `Fresnel` object to describe dielectric or conductor Fresnel properties and an additional `Spectrum` object, which is used to scale the reflected color.

⟨*SpecularReflection Public Methods*⟩ ≡ 440
```
SpecularReflection(const Spectrum &r, Fresnel *f)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_SPECULAR)),
      R(r), fresnel(f) {
}
```

⟨*SpecularReflection Private Data*⟩ ≡ 440
```
Spectrum R;
Fresnel *fresnel;
```

The rest of the implementation is straightforward. No scattering is returned from `SpecularReflection::f()`, since for an arbitrary pair of directions the delta function returns no scattering.[3]

3 If the caller happened to pass a vector and its perfect mirror direction, this function still returns zero. Although this might be a slightly confusing interface to these reflection functions, we still get the correct result in the end because reflection functions involving singularities with delta distributions receive special handling by the light transport routines (see Chapter 15).

BSDF_REFLECTION 428
BSDF_SPECULAR 428
BxDF 428
BxDFType 428
Fresnel 436
Spectrum 263
SpecularReflection 440
SpecularReflection::f() 441

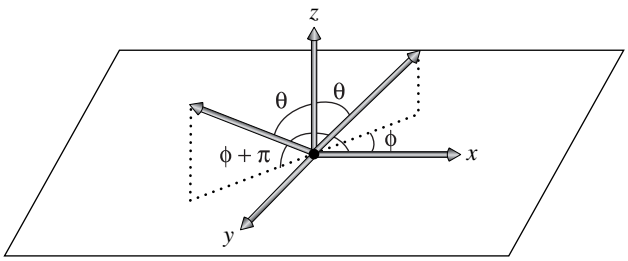


Figure 8.7: Given an incident direction that makes an angle θ with the surface normal and an angle ϕ with the x axis, the reflected ray about the normal makes an angle θ with the normal and $\phi + \pi$ with the x axis. The (x, y, z) coordinates of this direction can be found by scaling the incident direction by $(-1, -1, 1)$.

⟨*SpecularReflection Public Methods*⟩ +≡ 440
```
Spectrum f(const Vector &, const Vector &) const {
    return Spectrum(0.);
}
```

However, we do implement the `Sample_f()` method, which selects an appropriate direction according to the delta function. It sets the output variable `wi` to be the reflection of the supplied direction `wo` about the surface normal. The `*pdf` value is set to be one, which is the appropriate value for this case, where no Monte Carlo sampling is being done.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum SpecularReflection::Sample_f(const Vector &wo,
        Vector *wi, float u1, float u2, float *pdf) const {
    ⟨Compute perfect specular reflection direction 441⟩
    *pdf = 1.f;
    return fresnel->Evaluate(CosTheta(wo)) * R / AbsCosTheta(*wi);
}
```

The desired direction is the reflection of ω_o around the surface normal. Because all computations take place in a shading coordinate system where the surface normal is (0, 0, 1), we just rotate ω_i by π radians about $\mathbf{n}$ (Figure 8.7). Recall the transformation matrix from Chapter 2 for a rotation around the z axis; if the angle of rotation is π radians, the matrix is

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

When a vector is multiplied by this matrix, the effect is just to negate the x and y components.

⟨*Compute perfect specular reflection direction*⟩ ≡ 441
```
*wi = Vector(-wo.x, -wo.y, wo.z);
```

CosTheta() 426
Fresnel::Evaluate() 436
Spectrum 263
SpecularReflection 440
SpecularReflection::R 440
Vector 57

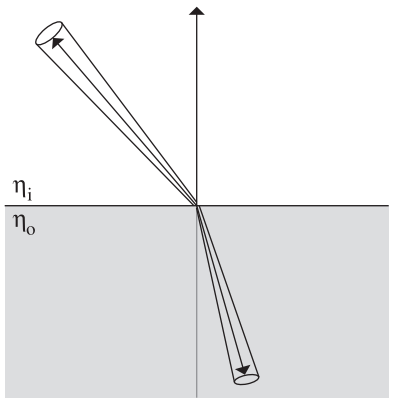


Figure 8.8: The amount of transmitted radiance at the boundary between media with different indices of refraction is scaled by the squared ratio of the two indices of refraction. Intuitively, this can be understood as the result of the radiance's differential solid angle being compressed or expanded as a result of transmission.

8.2.3 SPECULAR TRANSMISSION

We will now derive the BTDF for specular transmission. Snell's law is the basis of the derivation. Not only does it give the direction for the transmitted ray, but it can also be used to show that radiance along a ray changes as the ray goes between media with different indices of refraction.

Consider incident radiance arriving at the boundary between two media, with indices of refraction η_i and η_o for the incoming and outgoing media, respectively (Figure 8.8). We use τ to denote the fraction of incident energy that is transmitted to the outgoing direction, as given by the Fresnel equations, so $\tau = 1 - F_r(\omega_i)$. The amount of transmitted differential flux, then, is

$$\mathrm{d}\Phi_o = \tau \mathrm{d}\Phi_i.$$

If we use the definition of radiance, Equation (5.2), we have

$$(L_o \cos\theta_o \,\mathrm{d}A\, \mathrm{d}\omega_o) = \tau (L_i \cos\theta_i \,\mathrm{d}A\, \mathrm{d}\omega_i).$$

Expanding the solid angles to spherical angles, we have

$$(L_o \cos\theta_o \,\mathrm{d}A \sin\theta_o \,\mathrm{d}\theta_o \,\mathrm{d}\phi_o) = \tau (L_i \cos\theta_i \,\mathrm{d}A \sin\theta_i \,\mathrm{d}\theta_i \,\mathrm{d}\phi_i). \quad [8.4]$$

We can now differentiate Snell's law with respect to θ, which gives the relation

$$\eta_o \cos\theta_o \,\mathrm{d}\theta_o = \eta_i \cos\theta_i \,\mathrm{d}\theta_i.$$

Rearranging terms, we get

$$\frac{\cos\theta_o \,\mathrm{d}\theta_o}{\cos\theta_i \,\mathrm{d}\theta_i} = \frac{\eta_i}{\eta_o}.$$

Substituting this and Snell's law into Equation (8.4) and simplifying, we have

$$L_o \, \eta_i^2 \, d\phi_o = \tau L_i \, \eta_o^2 \, d\phi_i.$$

Because $\phi_i = \phi_o + \pi$ and therefore $d\phi_i = d\phi_o$, this gives the final relationship:

$$L_o = \tau L_i \frac{\eta_o^2}{\eta_i^2}. \tag{8.5}$$

The BTDF for specular transmission is thus

$$f_t(p, \omega_i, \omega_o) = \frac{\eta_o^2}{\eta_i^2}(1 - F_r(\omega_i)) \frac{\delta(\omega_i - T(\omega_o, \mathbf{n}))}{|\cos \theta_i|},$$

where $T(\omega_o, \mathbf{n})$ is the specular transmission vector for ω_o about the surface normal $\mathbf{n}$. The $1 - F_r(\omega_i)$ term in this equation corresponds to an easily observed effect: transmission is stronger at near-perpendicular angles. For example, if you look straight down into a clear lake, you can see far into the water, but at grazing angles most of the light is reflected as if from a mirror.

The `SpecularTransmission` class is almost exactly the same as `SpecularReflection` except that the sampled direction is respectively the direction for perfect specular transmission. Figure 8.9 shows an image of the Killeroo model using specular reflection and transmission BRDF and BTDF to model glass.

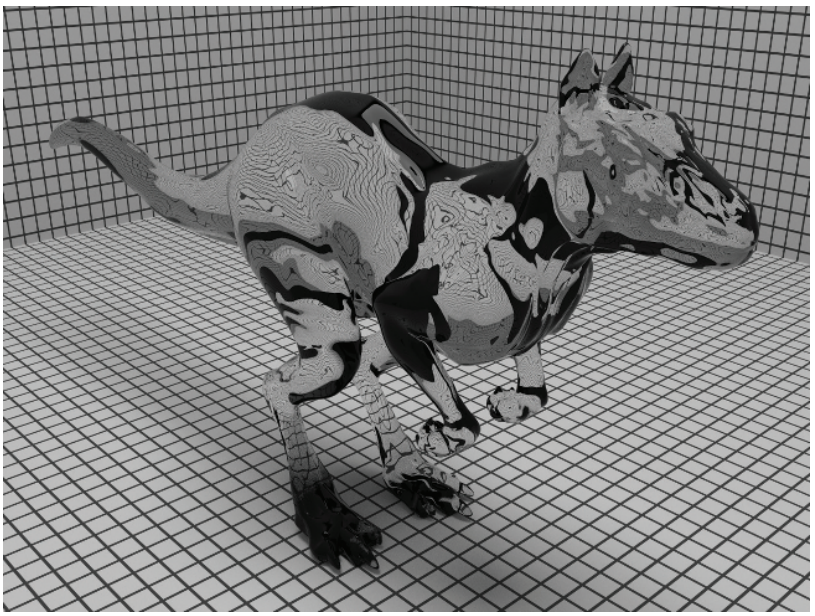


Figure 8.9: When the BRDF for specular reflection and the BTDF for specular transmission are modulated with the Fresnel formula for dielectrics, the realistic angle-dependent variation of the amount of reflection and transmission makes the result more visually convincing. *(Model courtesy of headus/Rezard.)*

SpecularReflection 440
SpecularTransmission 444

⟨*BxDF Declarations*⟩ +≡
```
class SpecularTransmission : public BxDF {
public:
    ⟨SpecularTransmission Public Methods 444⟩
private:
    ⟨SpecularTransmission Private Data 444⟩
};
```

The `SpecularTransmission` constructor stores the two indices of refraction on either side of the surface, as well as a transmission scale factor `T`.

⟨*SpecularTransmission Public Methods*⟩ ≡ 444
```
SpecularTransmission(const Spectrum &t, float ei, float et)
    : BxDF(BxDFType(BSDF_TRANSMISSION | BSDF_SPECULAR)),
      fresnel(ei, et) {
    T = t;
    etai = ei;
    etat = et;
}
```

Because conductors do not transmit light, a `FresnelDielectric` object is always used to do the Fresnel computations.

⟨*SpecularTransmission Private Data*⟩ ≡ 444
```
Spectrum T;
float etai, etat;
FresnelDielectric fresnel;
```

As with `SpecularReflection`, zero is always returned from `SpecularTransmission::f()`, since the BTDF is a scaled delta distribution.

⟨*SpecularTransmission Public Methods*⟩ +≡ 444
```
Spectrum f(const Vector &, const Vector &) const {
    return Spectrum(0.);
}
```

Figure 8.10 shows the geometry of specular transmission. The incident ray is refracted about the surface normal, with the angle θ_t given by Snell's law.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum SpecularTransmission::Sample_f(const Vector &wo,
        Vector *wi, float u1, float u2, float *pdf) const {
    ⟨Figure out which η is incident and which is transmitted 445⟩
    ⟨Compute transmitted ray direction 445⟩
    *pdf = 1.f;
    Spectrum F = fresnel.Evaluate(CosTheta(wo));
    return (et*et)/(ei*ei) * (Spectrum(1.)-F) * T /
        AbsCosTheta(*wi);
}
```

BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDF 428
BxDFType 428
CosTheta() 426
Fresnel::Evaluate() 436
FresnelDielectric 437
Spectrum 263
SpecularReflection 440
SpecularTransmission 444
SpecularTransmission::etai 444
SpecularTransmission::etat 444
SpecularTransmission::f() 444
SpecularTransmission::T 444
Vector 57

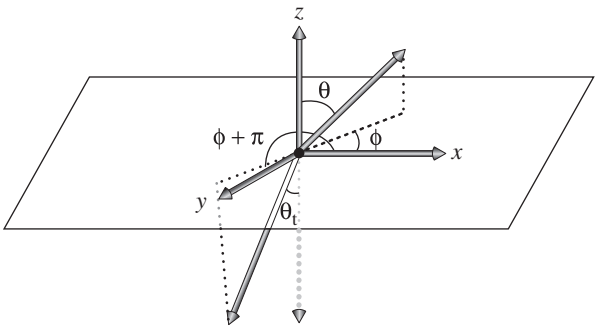


Figure 8.10: The Geometry of Specular Transmission. The specularly transmitted direction makes an angle θ_t with the surface normal. Like specular reflection, the angle it makes with the x axis is π greater than the incident ray's angle.

The method first determines whether the incident ray is entering or exiting the refractive medium. We use the convention that the surface normal, and thus the (0, 0, 1) direction in local reflection space, is oriented such that it points toward the outside of the object. Therefore, if the z component of the ω_o direction is greater than zero, the incident ray is coming from outside of the object.

⟨*Figure out which η is incident and which is transmitted*⟩ ≡ 444
```
bool entering = CosTheta(wo) > 0.;
float ei = etai, et = etat;
if (!entering)
    swap(ei, et);
```

Figure 8.11 shows the basic setting for computing the transmitted ray direction.

We next compute `sini2` and `sint2`, which are the squares of $\sin\theta_i$ and $\sin\theta_t$, respectively. In the reflection coordinate system, $\sin\theta_i$ is equal to the sum of the squares of the x and y components of ω_o; $\sin^2\theta_t$ can be computed directly from $(\sin\theta_i)^2$ using Snell's law. (Note the overloaded notation: these values are for the incident and transmitted media, though here the incident medium is the one the outgoing direction ω_o lies in.)

We then apply the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$ to compute $\cos\theta_t$ from $\sin\theta_t$; this directly gives us the z component of the transmitted direction. To compute the x and y components, we first mirror ω_o about the normal, as we did for specular reflection, but then scale it by the ratio $\sin\theta_t / \sin\theta_i$ to give it the proper magnitude. From Snell's law, this ratio is just η_i/η_t, which we happen to have computed previously.

CosTheta() 426
SinTheta2() 426
SpecularTransmission::
 etai 444
SpecularTransmission::
 etat 444
Vector 57

⟨*Compute transmitted ray direction*⟩ ≡ 444
```
float sini2 = SinTheta2(wo);
float eta = ei / et;
float sint2 = eta * eta * sini2;
⟨Handle total internal reflection for transmission 446⟩
```

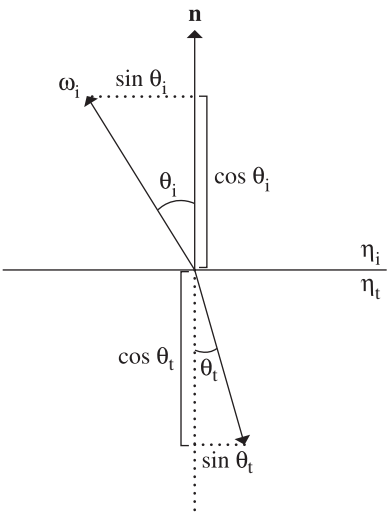


Figure 8.11: Geometry for computing the transmitted direction ω_t from the incident direction ω_i. The $\cos\theta$ terms are equal to the z components of the corresponding direction vectors, and the $\sin\theta$ terms are equal to the xy lengths of the direction vectors.

```
float cost = sqrtf(max(0.f, 1.f - sint2));
if (entering) cost = -cost;
float sintOverSini = eta;
*wi = Vector(sintOverSini * -wo.x, sintOverSini * -wo.y, cost);
```

We need to handle the case of total internal reflection here as well. If the squared value of $\sin\theta_t$ is greater than or equal to one, no transmission is possible, so black is returned.[4]

⟨*Handle total internal reflection for transmission*⟩ ≡ 445

```
if (sint2 >= 1.) return 0.;
```

8.3 LAMBERTIAN REFLECTION

One of the simplest BRDFs is the Lambertian model. It models a perfect diffuse surface that scatters incident illumination equally in all directions. Although this reflection model is not physically plausible, it is a good approximation to many real-world surfaces such as matte paint.

4 The first version of `pbrt` had a test > 1 rather than ≥ 1. Though the difference between the two may seem innocuous, this discrepancy led to not-a-number values occasionally being computed due to the z component of ω_i being zero (in the tangent plane of the surface) and thus to the $1/\cos\theta$ term being infinite.

⟨*BxDF Declarations*⟩ +≡
```
class Lambertian : public BxDF {
public:
    ⟨Lambertian Public Methods 447⟩
private:
    ⟨Lambertian Private Data 447⟩
};
```

The `Lambertian` constructor takes a reflectance spectrum R, which gives the fraction of incident light that is scattered.

⟨*Lambertian Public Methods*⟩ ≡ 447
```
Lambertian(const Spectrum &reflectance)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)), R(reflectance) { }
```

⟨*Lambertian Private Data*⟩ ≡ 447
```
Spectrum R;
```

The reflection distribution function for `Lambertian` is quite straightforward, since its value is constant. However, the value R/π must be returned, rather than R: the constructor takes the BRDF's reflectance; equating this to the earlier ρ_{hd} integral and solving for the BRDF's value demonstrates why this is the correct value to return for the BRDF.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum Lambertian::f(const Vector &wo, const Vector &wi) const {
    return R * INV_PI;
}
```

The directional-hemispherical and hemispherical-hemispherical reflectance values for a Lambertian BRDF are trivial to compute analytically, so the derivations are omitted here.

⟨*Lambertian Public Methods*⟩ +≡ 447
```
Spectrum rho(const Vector &, int, const float *) const { return R; }
Spectrum rho(int, const float *, const float *) const { return R; }
```

8.4 MICROFACET MODELS

Many geometric-optics-based approaches to modeling surface reflection are based on the idea that rough surfaces can be modeled as a collection of small *microfacets*. A surface comprised of microfacets is essentially a heightfield, where the distribution of facets is described statistically. Figure 8.12 shows cross sections of a relatively rough surface and a much smoother microfacet surface.

Microfacet-based BRDF models work by statistically modeling the scattering of light from a large collection of microfacets. If we assume that the differential area being illuminated, $\mathrm{d}A$, is relatively large compared to the size of individual microfacets, then a large number of microfacets are illuminated, and so their aggregate behavior determines the scattering.

BSDF_DIFFUSE 428
BSDF_REFLECTION 428
BxDF 428
BxDFType 428
INV_PI 1002
Lambertian 447
Lambertian::R 447
Spectrum 263
Vector 57

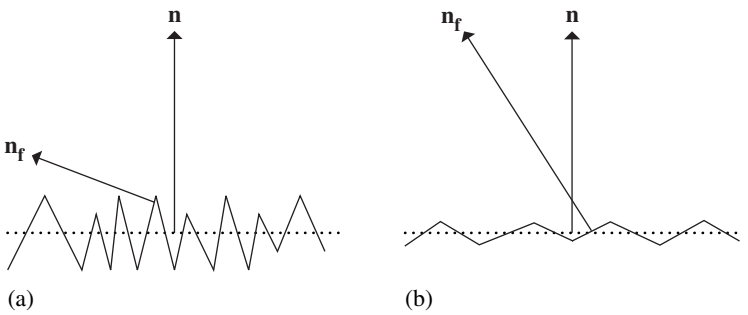


Figure 8.12: Microfacet surface models are often described by a function that gives the distribution of microfacet normals $\mathbf{n}_f$ with respect to the surface normal $\mathbf{n}$. (a) The greater the variation of microfacet normals, the rougher the surface is. (b) Smooth surfaces have relatively little variation of microfacet normals.

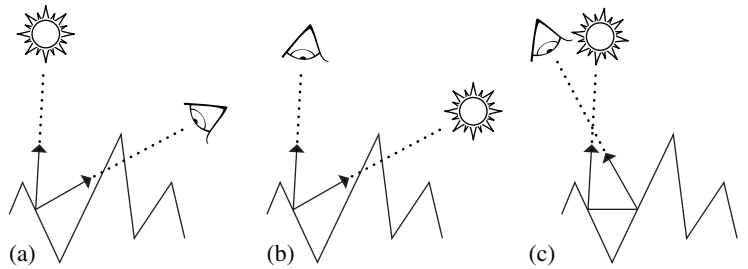


Figure 8.13: Three Important Geometric Effects to Consider with Microfacet Reflection Models. (a) *Masking*: the microfacet of interest isn't visible to the viewer due to occlusion by another microfacet. (b) *Shadowing*: analogously, light doesn't reach the microfacet. (c) *Interreflection*: light bounces among the microfacets before reaching the viewer.

The two main components of microfacet models are an expression for the distribution of facets and a BRDF that describes how light scatters from individual microfacets. Given these, the goal is to derive a closed-form expression giving the BRDF that describes scattering from such a surface. Perfect mirror reflection is typically used for the microfacet BRDF, although the Oren–Nayar model (described in the next section) treats them as Lambertian reflectors.

To compute reflection from such a model, local lighting effects at the microfacet level need to be considered (Figure 8.13). Microfacets may be occluded by another facet, may lie in the shadow of a neighboring microfacet, or interreflection may cause a microfacet to reflect more light than predicted by the amount of direct illumination and the low-level microfacet BRDF. A common simplification is to assume that all of the microfacets make up symmetric V-shaped grooves. If this assumption is made, then interreflection

with most of the other microfacets can be ignored and only the neighboring microfacet needs to be considered.

Particular microfacet-based BRDF models consider each of these effects with varying degrees of accuracy. The general approach is to make the best approximations possible, while still obtaining an easily evaluated expression.

8.4.1 OREN–NAYAR DIFFUSE REFLECTION

Oren and Nayar (1994) observed that real-world objects tend not to exhibit perfect Lambertian reflection. Specifically, rough surfaces generally appear brighter as the illumination direction approaches the viewing direction. They developed a reflection model that describes rough surfaces as a collection of symmetric V-shaped grooves in an effort to better model effects like these. They further assumed that each individual microfacet (groove face) exhibited perfect Lambertian reflection and derived a BRDF that models the aggregate reflection of the collection of grooves. The distribution of microfacets was modeled with a Gaussian distribution with a single parameter σ, the standard deviation of the orientation angle.

The resulting model, which accounts for shadowing, masking, and interreflection among the microfacets, does not have a closed-form solution, so they found the following approximation that fit it well:

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}\left(A + B\max(0, \cos(\phi_i - \phi_o))\sin\alpha\tan\beta\right),$$

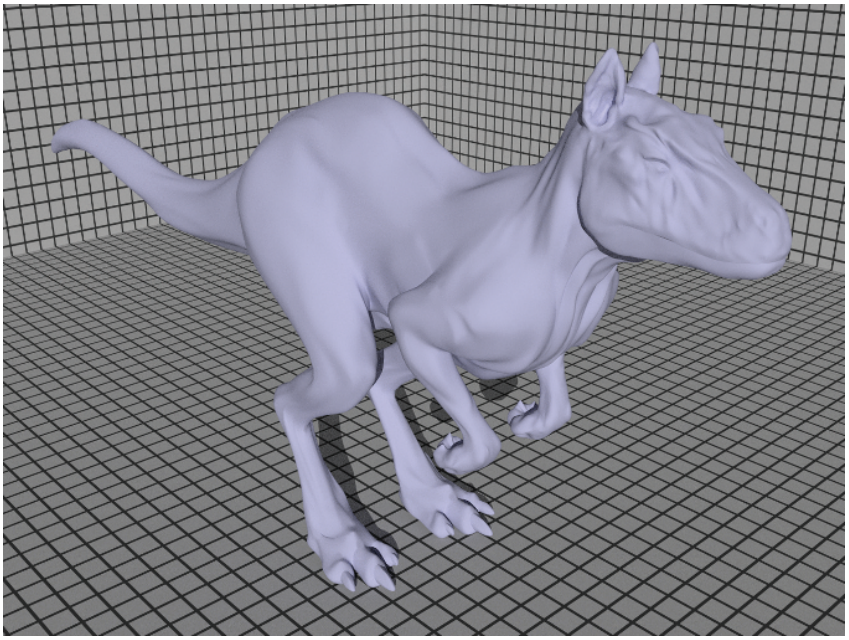
where if σ is in radians,

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)}$$
$$B = \frac{0.45\sigma^2}{\sigma^2 + 0.09}$$
$$\alpha = \max(\theta_i, \theta_o)$$
$$\beta = \min(\theta_i, \theta_o).$$

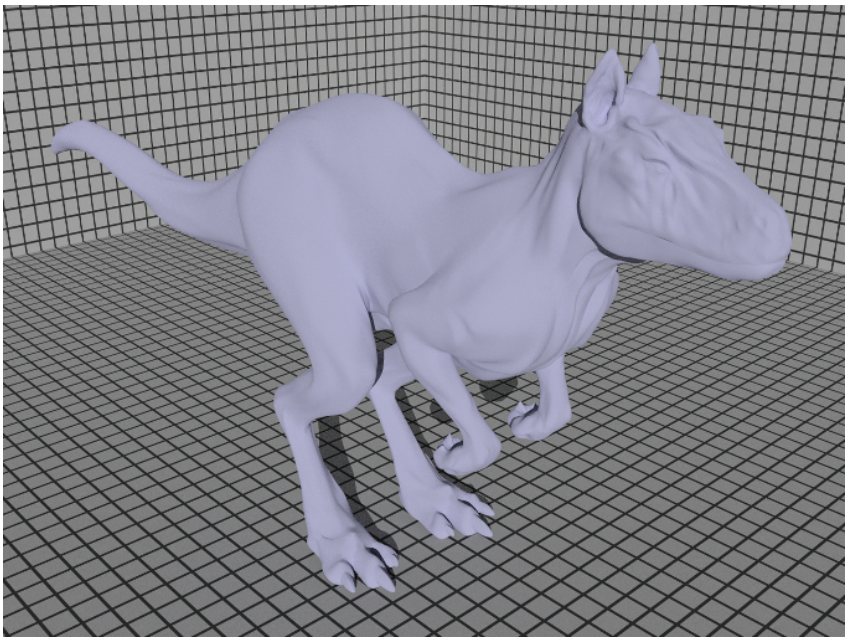
The implementation precomputes and stores the values of the A and B parameters in the constructor to save work in evaluating the BRDF later. Figure 8.14 compares the difference between rendering with ideal diffuse reflection and with the Oren–Nayar model.

BSDF_DIFFUSE 428
BSDF_REFLECTION 428
BxDF 428
BxDFType 428
OrenNayar::A 451
OrenNayar::B 451
OrenNayar::R 451
Spectrum 263

⟨*OrenNayar Public Methods*⟩ ≡
```
OrenNayar(const Spectrum &reflectance, float sig)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)),
      R(reflectance) {
    float sigma = Radians(sig);
    float sigma2 = sigma*sigma;
    A = 1.f - (sigma2 / (2.f * (sigma2 + 0.33f)));
    B = 0.45f * sigma2 / (sigma2 + 0.09f);
}
```



(a)



(b)

Figure 8.14: Killeroo model rendered (a) with standard diffuse reflection from the `Lambertian` model and (b) with the `OrenNayar` model with a σ parameter of 20 degrees. Note the increase in reflection at the silhouette edges and the generally less-drawn-out transitions at light terminator edges with the Oren–Nayar model. *(Model courtesy of headus/Rezard.)*

⟨*OrenNayar Private Data*⟩ ≡
```
Spectrum R;
float A, B;
```

Application of trigonometry can substantially improve the efficiency of the evaluation routine compared to a naive implementation. The implementation starts by computing the values of $\sin\theta_i$ and $\sin\theta_o$.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum OrenNayar::f(const Vector &wo, const Vector &wi) const {
    float sinthetai = SinTheta(wi);
    float sinthetao = SinTheta(wo);
    ⟨Compute cosine term of Oren-Nayar model 451⟩
    ⟨Compute sine and tangent terms of Oren-Nayar model 451⟩
    return R * INV_PI * (A + B * maxcos * sinalpha * tanbeta);
}
```

It is now necessary to compute the $\max(0, \cos(\phi_i - \phi_o))$ term. We can apply the trigonometric identity

$$\cos(a - b) = \cos a \cos b + \sin a \sin b,$$

such that we just need to compute the sines and cosines of ϕ_i and ϕ_o.

⟨*Compute cosine term of Oren-Nayar model*⟩ ≡ 451
```
float maxcos = 0.f;
if (sinthetai > 1e-4 && sinthetao > 1e-4) {
    float sinphii = SinPhi(wi), cosphii = CosPhi(wi);
    float sinphio = SinPhi(wo), cosphio = CosPhi(wo);
    float dcos = cosphii * cosphio + sinphii * sinphio;
    maxcos = max(0.f, dcos);
}
```

Finally, the $\sin\alpha$ and $\tan\beta$ terms are found. Note that whichever of ω_i or ω_o has a larger value for $\cos\theta$ (i.e., a larger z component) has a *smaller* value for θ. We can set $\sin\alpha$ using the appropriate sine value computed at the beginning of the method. The tangent can then be computed using the identity $\tan a = \sin a / \cos a$.

AbsCosTheta() 426
CosPhi() 427
INV_PI 1002
OrenNayar 449
OrenNayar::A 451
OrenNayar::B 451
OrenNayar::R 451
SinPhi() 427
SinTheta() 426
Spectrum 263
Vector 57

⟨*Compute sine and tangent terms of Oren-Nayar model*⟩ ≡ 451
```
float sinalpha, tanbeta;
if (AbsCosTheta(wi) > AbsCosTheta(wo)) {
    sinalpha = sinthetao;
    tanbeta = sinthetai / AbsCosTheta(wi);
}
else {
    sinalpha = sinthetai;
    tanbeta = sinthetao / AbsCosTheta(wo);
}
```

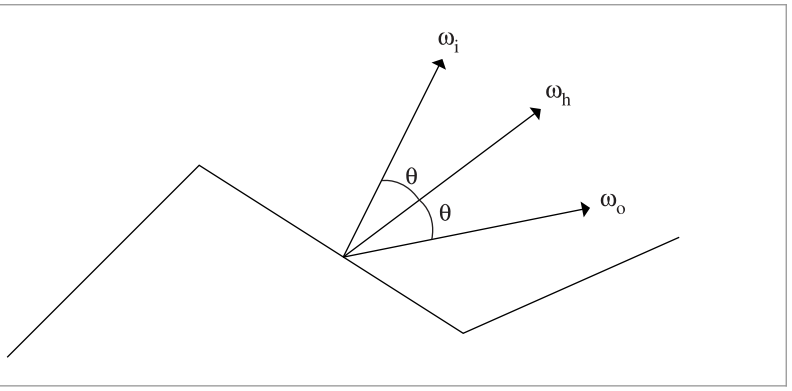


Figure 8.15: For perfectly specular microfacets and a given pair of directions ω_i and ω_o, only those microfacets with normal $\omega_h = \omega_i \widehat{+} \omega_o$ will reflect any light from ω_i to ω_o.

8.4.2 TORRANCE–SPARROW MODEL

One of the first microfacet models for computer graphics was developed by Torrance and Sparrow (1967) to model metallic surfaces. They modeled surfaces as collections of perfectly smooth mirrored microfacets. The surface is statistically described by a distribution function $D(\omega_h)$ that gives the probability that a microfacet has orientation ω_h (recall Figure 8.12, which shows how roughness and the microfacet normal distribution function are related).

Because the microfacets are perfectly specular, only those with a normal equal to the *half-angle vector*,

$$\omega_h = \omega_i \widehat{+} \omega_o,$$

cause perfect specular reflection from ω_i to ω_o (Figure 8.15).

The derivation of the Torrance–Sparrow model has a number of interesting steps; we'll go through it in detail here. Consider the differential flux $d\Phi_h$ incident on the microfacets oriented with half-angle ω_h for directions ω_i and ω_o. From the definition of radiance, Equation (5.2), it is

$$d\Phi_h = L_i(\omega_i)\, d\omega\, dA^{\perp}(\omega_h) = L_i(\omega_i)\, d\omega\, \cos\theta_h\, dA(\omega_h),$$

where we have written $dA(\omega_h)$ for the area measure of the microfacets with orientation ω_h and $\cos\theta_h$ for the cosine of the angle between ω_i and ω_h (Figure 8.16).

The differential area of microfacets with orientation ω_h is

$$dA(\omega_h) = D(\omega_h)\, d\omega_h\, dA.$$

The first two terms describe the differential area of facets per unit area that have the proper orientation, and the dA term converts this to differential area.

Therefore,

$$d\Phi_h = L_i(\omega_i)\, d\omega\, \cos\theta_h\, D(\omega_h)\, d\omega_h\, dA. \quad \text{(8.6)}$$

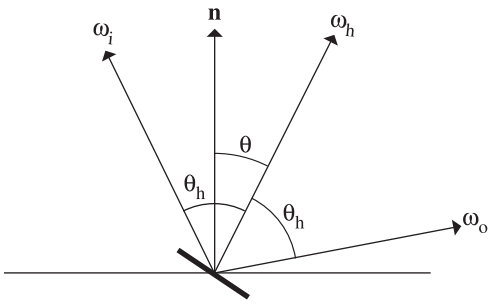


Figure 8.16: Setting for the Derivation of the Torrance–Sparrow Model. For directions ω_i and ω_o, only microfacets with normal ω_h reflect light. The angle between ω_h and $\mathbf{n}$ is denoted by θ, and the angle between ω_h and ω_o is denoted by θ_h. (The angle between ω_h and ω_i is also necessarily θ_h.)

If we assume that the microfacets individually reflect light according to Fresnel's law, the outgoing flux is

$$d\Phi_o = F_r(\omega_o) d\Phi_h. \quad [8.7]$$

Again using the definition of radiance, the reflected outgoing radiance is

$$L(\omega_o) = \frac{d\Phi_o}{d\omega_o \cos\theta_o \, dA}.$$

If we substitute Equation (8.7) into this and then Equation (8.6) into the result, we have

$$L(\omega_o) = \frac{F_r(\omega_o) L_i(\omega_i) \, d\omega_i \, D(\omega_h) \, d\omega_h \, dA \cos\theta_h}{d\omega_o \, dA \cos\theta_o}.$$

In Section 14.5.1, we will derive an important relationship between $d\omega_h$ and $d\omega_o$:

$$d\omega_h = \frac{d\omega_o}{4 \cos\theta_h}.$$

We can substitute this into the previous equation and simplify, giving

$$L(\omega_o) = \frac{F_r(\omega_o) L_i(\omega_i) D(\omega_h) \, d\omega_i}{4 \cos\theta_o}.$$

We can now apply the definition of the BRDF, Equation (5.7), giving us the Torrance–Sparrow BRDF:

$$f_r(p, \omega_o, \omega_i) = \frac{D(\omega_h) F_r(\omega_o)}{4 \cos\theta_o \cos\theta_i}.$$

The Torrance–Sparrow model also includes a *geometric attenuation* term, which describes the fraction of microfacets that are masked or shadowed, given directions ω_i and ω_o. This

G term can just be included in the derivation as the Fresnel term was earlier. The full model, then, is

$$f_r(p, \omega_o, \omega_i) = \frac{D(\omega_h)\, G(\omega_o, \omega_i)\, F_r(\omega_o)}{4 \cos\theta_o \cos\theta_i}. \quad [8.8]$$

One of the nice things about the Torrance–Sparrow model is that the derivation doesn't depend on the particular microfacet distribution being used. Furthermore, it doesn't depend on a particular Fresnel function, so it can be used for both conductors and dielectrics. However, reflection functions other than perfect specular reflection cannot be easily substituted: the relationship between $d\omega_h$ and $d\omega_o$ used in the derivation depends on the specular reflection assumption.

We now use the Torrance–Sparrow model to implement a general microfacet-based BRDF. It takes a pointer to an abstract `MicrofacetDistribution` class, which provides a single method to compute the D term of the Torrance–Sparrow model. This function, `MicrofacetDistribution::D()`, gives the probability density for microfacets to be oriented with normal ω_h.

⟨*BxDF Declarations*⟩ +≡
```
class MicrofacetDistribution {
public:
    ⟨MicrofacetDistribution Interface 454⟩
};
```

⟨*MicrofacetDistribution Interface*⟩ ≡ 454
```
virtual float D(const Vector &wh) const = 0;
```

The `Microfacet` BRDF, then, just takes a reflectance, the Fresnel function, and a pointer to a distribution.

⟨*BxDF Declarations*⟩ +≡
```
class Microfacet : public BxDF {
public:
    ⟨Microfacet Public Methods 455⟩
private:
    ⟨Microfacet Private Data 454⟩
};
```

⟨*BxDF Method Definitions*⟩ +≡
```
Microfacet::Microfacet(const Spectrum &reflectance, Fresnel *f,
                       MicrofacetDistribution *d)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)),
      R(reflectance), distribution(d), fresnel(f) {
}
```

⟨*Microfacet Private Data*⟩ ≡ 454
```
Spectrum R;
MicrofacetDistribution *distribution;
Fresnel *fresnel;
```

BSDF_GLOSSY 428
BSDF_REFLECTION 428
BxDF 428
BxDFType 428
Fresnel 436
Microfacet 454
Microfacet::distribution 454
Microfacet::fresnel 454
Microfacet::R 454
MicrofacetDistribution 454
MicrofacetDistribution::D() 454
Spectrum 263
Vector 57

Evaluating the terms of the Torrance–Sparrow BRDF is straightforward. For the Fresnel term, recall that the angle θ_h is the same between ω_h and both ω_i and ω_o, so it doesn't matter which vector we use to compute the cosine of θ_h. We arbitrarily choose ω_i.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum Microfacet::f(const Vector &wo, const Vector &wi) const {
    float cosThetaO = AbsCosTheta(wo);
    float cosThetaI = AbsCosTheta(wi);
    if (cosThetaI == 0.f || cosThetaO == 0.f) return Spectrum(0.f);
    Vector wh = Normalize(wi + wo);
    float cosThetaH = Dot(wi, wh);
    Spectrum F = fresnel->Evaluate(cosThetaH);
    return R * distribution->D(wh) * G(wo, wi, wh) * F /
               (4.f * cosThetaI * cosThetaO);
}
```

The geometric attenuation term is derived by assuming that the microfacets are arranged along infinitely long V-shaped grooves. This assumption is more restrictive than the general term $D(\omega_h)$ used to model the microfacet distribution and does not account for the roughness of the surface, but yields a closed-form result, which is helpful for efficiency. It is also easy to evaluate and matches many real-world surfaces well. The attenuation term (omitting the derivation) is

$$G(\omega_o, \omega_i) = \min\left(1, \min\left(\frac{2(\mathbf{n}\cdot\omega_h)(\mathbf{n}\cdot\omega_o)}{\omega_o\cdot\omega_h}, \frac{2(\mathbf{n}\cdot\omega_h)(\mathbf{n}\cdot\omega_i)}{\omega_o\cdot\omega_h}\right)\right).$$

⟨*Microfacet Public Methods*⟩ ≡ 454
```
float G(const Vector &wo, const Vector &wi, const Vector &wh) const {
    float NdotWh = AbsCosTheta(wh);
    float NdotWo = AbsCosTheta(wo);
    float NdotWi = AbsCosTheta(wi);
    float WOdotWh = AbsDot(wo, wh);
    return min(1.f, min((2.f * NdotWh * NdotWo / WOdotWh),
                        (2.f * NdotWh * NdotWi / WOdotWh)));
}
```

AbsCosTheta() 426
AbsDot() 61
Dot() 60
Fresnel::Evaluate() 436
Microfacet 454
Microfacet::G() 455
Microfacet::R 454
MicrofacetDistribution::D() 454
Spectrum 263
Vector 57
Vector::Normalize() 63

8.4.3 BLINN MICROFACET DISTRIBUTION

Blinn (1977) proposed a model where the distribution of microfacet normals is approximated by an exponential falloff. The most likely microfacet orientation in this model is in the surface normal direction, falling off to no microfacets oriented perpendicular to the normal. For smooth surfaces, this falloff happens very quickly; for rough surfaces, it is more gradual.

The Blinn model raises the cosine of the angle between the half-vector and the normal to an exponent e:

$$D(\omega_h) \propto (\omega_h \cdot \mathbf{n})^e.$$

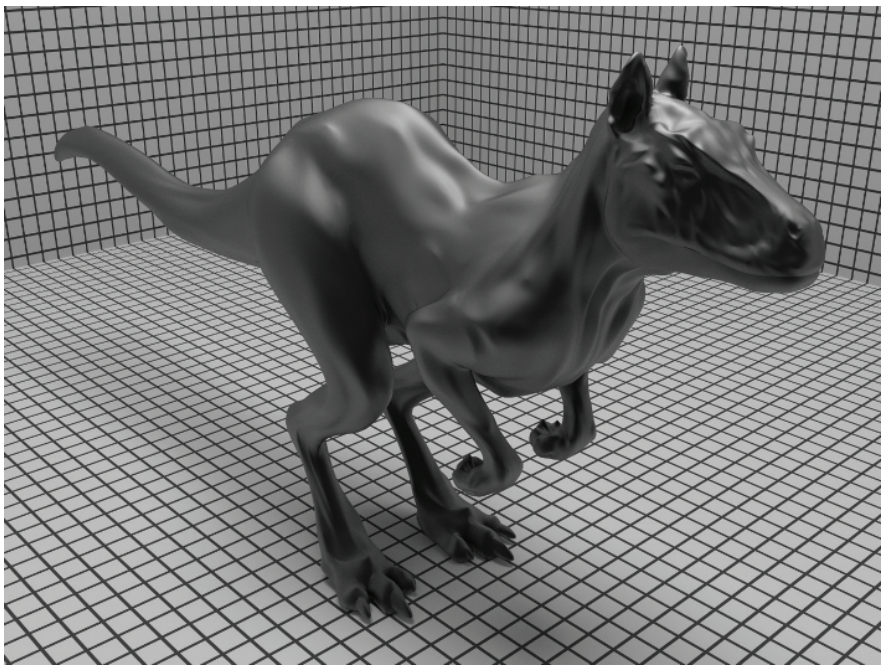


Figure 8.17: Killeroo model rendered with the Torrance–Sparrow microfacet model and Blinn microfacet distribution function. *(Model courtesy of headus/Rezard.)*

Figure 8.17 shows the Killeroo rendered with the Torrance–Sparrow model and the Blinn microfacet distribution.

⟨*BxDF Declarations*⟩ +≡
```
class Blinn : public MicrofacetDistribution {
public:
    Blinn(float e) { if (e > 10000.f || isnan(e)) e = 10000.f;
                     exponent = e; }
    ⟨Blinn Public Methods 457⟩
private:
    float exponent;
};
```

Microfacet distribution functions must be *normalized* to ensure that they are physically plausible. In the case of microfacets, there must exist some heightfield with the distribution of face normals $D(\omega_h)$. Another way to think about this restriction is that if we sum the projected area of all the microfacet faces over some area dA, the sum should equal dA. Mathematically, this means we must enforce

$$\int_{\mathcal{H}^2(\mathbf{n})} D(\omega_h) \cos\theta_h \, d\omega_h = 1.$$

Blinn 456
MicrofacetDistribution 454

A common error in normalizing microfacet distributions is to perform this integral over solid angle instead of projected solid angle (i.e., to leave out the $\cos\theta_h$ term), which does not guarantee the existence of a heightfield with the right distribution.

For the Blinn model, we know that $D(\omega_h) \propto (\omega_h \cdot \mathbf{n})^e$. The earlier normalization requirement gives

$$\begin{aligned}\int_{\mathcal{H}^2(\mathbf{n})} c(\omega_h \cdot \mathbf{n})^e \cos\theta_h \, d\omega_h &= \int_0^{2\pi} \int_0^{\frac{\pi}{2}} c(\cos\theta_h)^{e+1} \sin\theta_h \, d\theta \, d\phi \\ &= 2c\pi \int_0^1 u^{e+1} du \\ &= 2c\pi \left. \frac{u^{e+2}}{e+2} \right|_0^1 \\ &= \frac{2c\pi}{e+2} = 1.\end{aligned}$$

Therefore,

$$c = \frac{e+2}{2\pi},$$

and the properly normalized Blinn microfacet distribution is

$$D(\omega_h) = \frac{e+2}{2\pi}(\omega_h \cdot \mathbf{n})^e. \tag{8.9}$$

Figure 8.18 shows how the exponent affects the distribution. Figure 8.18(a) shows a plot of the Blinn model's distribution with outgoing direction shown and an exponent of 4. This corresponds to a fairly rough surface, so there is a high probability of microfacets being oriented in a direction far away from the normal. Figure 8.18(b) shows the distribution from an exponent of 20, corresponding to a smoother surface. For this case, there is a much lower probability that any microfacets will be oriented very far from the surface normal direction.

⟨*Blinn Public Methods*⟩ ≡ 456

```
float D(const Vector &wh) const {
    float costhetah = AbsCosTheta(wh);
    return (exponent+2) * INV_TWOPI * powf(costhetah, exponent);
}
```

8.4.4 ANISOTROPIC MICROFACET MODEL

Because the Blinn distribution described in the last section only depends on the angle between the half-angle and the surface normal, it is *radially symmetric* and yields an isotropic BRDF. Ashikhmin and Shirley (2000, 2002) developed a microfacet distribution function for modeling the appearance of anisotropic surfaces. Recall that an anisotropic BRDF is one where the reflection characteristics at a point vary as the surface is rotated about that point in the plane perpendicular to the surface normal. Brushed metals and some types of fabric exhibit anisotropy.

AbsCosTheta() 426
INV_TWOPI 1002
Vector 57

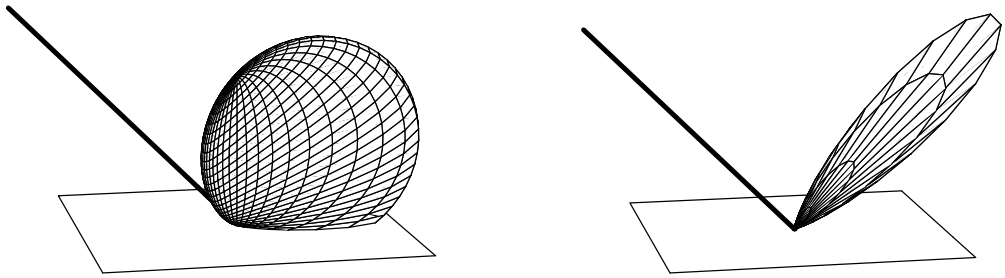


Figure 8.18: The Effect of Varying the Exponent for the Blinn Microfacet Distribution Model. (a) Distribution from exponent $e = 4$, (b) $e = 20$. The larger the exponent, the more likely it is that a microfacet will be oriented close to the surface normal, as would be the case for a smooth surface.

Ashikhmin and Shirley's model is physically based, has intuitive parameters, is efficient, and fits well into the Monte Carlo integration techniques that will be introduced in later chapters. We won't present the derivation here; the reader is referred to the "Further Reading" section for pointers to their original papers. Their model takes two parameters: e_x, which gives an exponent for the distribution function for half-angle vectors with an azimuthal angle that orients them exactly along the x axis, and e_y, an exponent for microfacets oriented along the y axis. Exponents for intermediate orientations are found by considering these two values as the lengths of the axes of an ellipse and finding the appropriate radius for the actual microfacet orientation (Figure 8.19).

The resulting microfacet distribution function is

$$D(\omega_h) = \frac{\sqrt{(e_x+2)(e_y+2)}}{2\pi}(\omega_h \cdot \mathbf{n})^{e_x \cos^2\phi + e_y \sin^2\phi}.$$

Figure 8.20 shows this model in use.

⟨*BxDF Declarations*⟩ +≡
```
class Anisotropic : public MicrofacetDistribution {
public:
    ⟨Anisotropic Public Methods 458⟩
private:
    float ex, ey;
};
```

⟨*Anisotropic Public Methods*⟩ ≡ 458
```
Anisotropic(float x, float y) {
    ex = x; ey = y;
    if (ex > 10000.f || isnan(ex)) ex = 10000.f;
    if (ey > 10000.f || isnan(ey)) ey = 10000.f;
}
```

Anisotropic 458
MicrofacetDistribution 454

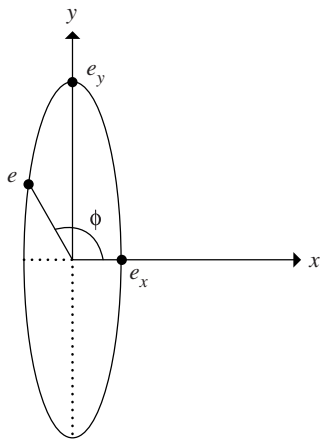


Figure 8.19: The two exponents e_x and e_y for the anisotropic microfacet distribution function give specular exponents for microfacets facing exactly along the x and y axes, respectively. For microfacets with other orientations, the exponent e is computed by finding the radius e of the ellipse for the actual orientation angle ϕ.

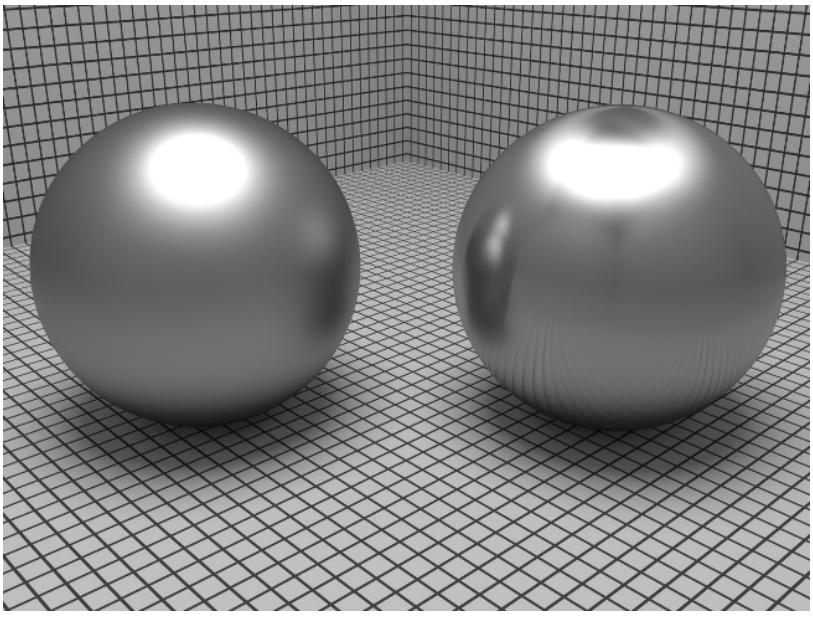


Figure 8.20: Spheres rendered with an isotropic microfacet distribution (left) and an anisotropic distribution (right). Note the different specular highlight shapes from the anisotropic model. We have used spheres here instead of the Killeroo, since anisotropic models like these depend on a globally consistent set of tangent vectors over the surface to orient the direction of anisotropy in a reasonable way.

The terms of the distribution function can be computed quite efficiently. Recall from the beginning of the chapter that $\cos\phi = x/\sin\theta$ and $\sin\phi = y/\sin\theta$. Since we want to compute $\cos^2\phi$ and $\sin^2\phi$, however, we can use the substitution $\sin^2\theta + \cos^2\theta = 1$, so that

$$\cos^2\phi = \frac{x^2}{1-z^2}$$

$$\sin^2\phi = \frac{y^2}{1-z^2}.$$

Thus, the implementation is

⟨*Anisotropic Public Methods*⟩ +≡ 458

```
float D(const Vector &wh) const {
    float costhetah = AbsCosTheta(wh);
    float d = 1.f - costhetah * costhetah;
    if (d == 0.f) return 0.f;
    float e = (ex * wh.x * wh.x + ey * wh.y * wh.y) / d;
    return sqrtf((ex+2.f) * (ey+2.f)) * INV_TWOPI * powf(costhetah, e);
}
```

8.5 FRESNEL INCIDENCE EFFECTS

Most BRDF models in graphics do not account for the fact that Fresnel reflection reduces the amount of light that reaches the bottom level of layered objects. Consider a polished wood table or a wall with glossy paint: if you look at their surfaces head-on, you primarily see the wood or the paint pigment color. As you move your viewpoint toward a glancing angle, you see less of the underlying color as it is overwhelmed by increasing glossy reflection due to Fresnel effects.

In this section, we will implement a BRDF model developed by Ashikhmin and Shirley (2000, 2002) that models a diffuse underlying surface with a glossy specular surface above it. The effect of reflection from the diffuse surface is modulated by the amount of energy left after Fresnel effects have been considered. Figure 8.21 shows this idea: When the incident direction is close to the normal, most light is transmitted to the diffuse layer and the diffuse term dominates. When the incident direction is close to glancing, glossy reflection is the primary mode of reflection. The car model in Figures 12.14 and 12.15 uses this BRDF for its paint.

⟨*BxDF Declarations*⟩ +≡

```
class FresnelBlend : public BxDF {
public:
    ⟨FresnelBlend Public Methods 462⟩
private:
    ⟨FresnelBlend Private Data 461⟩
};
```

The model takes two spectra, representing diffuse and specular reflectance, and a microfacet distribution for the glossy layer.

AbsCosTheta() 426
BxDF 428
INV_TWOPI 1002
Vector 57

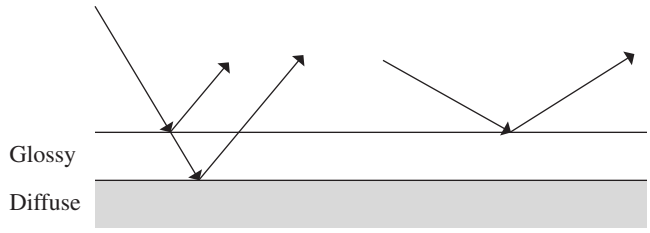


Figure 8.21: The `FresnelBlend` BRDF models the effect of a surface with a glossy layer on top of a diffuse substrate. As the angle of incidence of the vectors ω_i and ω_o heads toward glancing (right), the amount of light that reaches the diffuse substrate is reduced by Fresnel effects and the diffuse layer becomes less visibly apparent.

⟨*BxDF Method Definitions*⟩ +≡
```
FresnelBlend::FresnelBlend(const Spectrum &d, const Spectrum &s,
                           MicrofacetDistribution *dist)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)), Rd(d), Rs(s) {
    distribution = dist;
}
```

⟨*FresnelBlend Private Data*⟩ ≡ 460
```
Spectrum Rd, Rs;
MicrofacetDistribution *distribution;
```

This model is based on the weighted sum of a glossy specular term and a diffuse term. Accounting for reciprocity and energy conservation, the glossy specular term is derived as

$$f_r(p, \omega_o, \omega_i) = \frac{D(\omega_h)F(\omega_o)}{4\pi(\omega_h \cdot \omega_i)(\max((\mathbf{n} \cdot \omega_o), (\mathbf{n} \cdot \omega_i)))},$$

where $D(\omega_h)$ is a microfacet distribution term and $F(\omega_o)$ represents Fresnel reflectance. Note that this is quite similar to the Torrance–Sparrow model.

The key to Ashikhmin and Shirley's model is the derivation of a diffuse term such that the model still obeys reciprocity and conserves energy. The derivation is dependent on an approximation to the Fresnel reflection equations due to Schlick (1993), who approximated Fresnel reflection as

$$F_r(\cos\theta) = R + (1 - R)(1 - \cos\theta)^5,$$

where R is the reflectance of the surface at normal incidence.

Given this Fresnel term, the diffuse term in the following equation successfully models Fresnel-based reduced diffuse reflection in a physically plausible manner:

$$f_r(p, \omega_i, \omega_o) = \frac{28R_d}{23\pi}(1 - R_s)\left(1 - \left(1 - \frac{(\mathbf{n} \cdot \omega_i)}{2}\right)^5\right)\left(1 - \left(1 - \frac{(\mathbf{n} \cdot \omega_o)}{2}\right)^5\right).$$

We will not include the derivation of this result here.

BSDF_GLOSSY 428
BSDF_REFLECTION 428
BxDF 428
BxDFType 428
FresnelBlend 460
FresnelBlend::distribution 461
FresnelBlend::Rd 461
FresnelBlend::Rs 461
MicrofacetDistribution 454
Spectrum 263

⟨*FresnelBlend Public Methods*⟩ ≡ 460
```
Spectrum SchlickFresnel(float costheta) const {
    return Rs + powf(1 - costheta, 5.f) * (Spectrum(1.) - Rs);
}
```

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum FresnelBlend::f(const Vector &wo, const Vector &wi) const {
    Spectrum diffuse = (28.f/(23.f*M_PI)) * Rd *
        (Spectrum(1.f) - Rs) *
        (1.f - powf(1.f - .5f * AbsCosTheta(wi), 5)) *
        (1.f - powf(1.f - .5f * AbsCosTheta(wo), 5));
    Vector wh = Normalize(wi + wo);
    Spectrum specular = distribution->D(wh) /
        (4.f * AbsDot(wi, wh) * max(AbsCosTheta(wi), AbsCosTheta(wo))) *
        SchlickFresnel(Dot(wi, wh));
    return diffuse + specular;
}
```

8.6 MEASURED BRDFs

Using measured data about the reflection properties of real surfaces is one of the most effective approaches for rendering realistic materials. One way to do so is to use the measured data to set the parameter values for a parameterized BRDF like the Torrance–Sparrow model. If this can be done without introducing too much error, then doing so is worthwhile for the memory savings of not needing to store all of the measurement data in memory for rendering. (This parameter fitting would normally be done as a preprocess outside of the renderer.)

However, parameterized BRDF models are often not flexible enough to model the full complexity of scattering characteristics of many interesting surfaces. If accurate measured reflection data is available for a surface of interest, then directly using it for rendering can faithfully re-create the surface's appearance. In this section, we'll implement `BxDF`s that interpolate sampled BRDF values directly. While this is a memory-intensive approach for scenes that use many different measured BRDFs, it works well for reflection distributions that don't fit analytic models well and can provide a reference for comparison to other approaches. Figure 8.22 shows an image of the Killeroo model rendered with measured data from a red fabric.

Rendering with measured BRDF data does have challenges. It can be difficult to make adjustments for artistic purposes ("like that, just a little more shiny").[5] If there is excessive error in the data, results may be poor as well: especially for highly specular and anisotropic surfaces, the 4D BRDF must be densely sampled, leading to a large amount of data to store and many measurements to be taken. Finally, BRDFs may have considerable variation in their values, which adds to the challenges of acquiring accurate BRDF data sets.

AbsCosTheta() 426
BxDF 428
FresnelBlend 460
FresnelBlend::Rd 461
FresnelBlend::Rs 461
FresnelBlend::SchlickFresnel() 462
MicrofacetDistribution::D() 454
M_PI 1002
Spectrum 263
Vector 57
Vector::Normalize() 63

5 Though see the work of Matusik et al. (2003b) for an innovative approach to this problem.

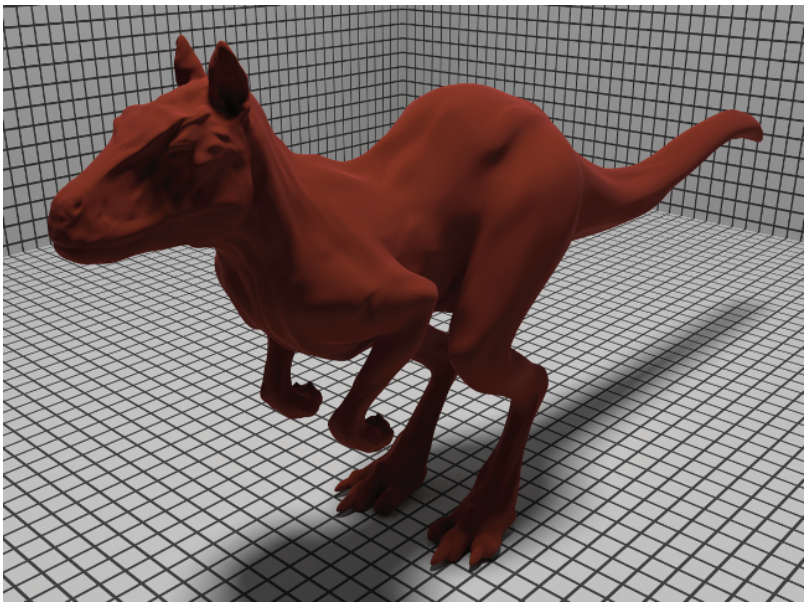


Figure 8.22: The Killeroo model rendered using measured red fabric BRDF data.

Measured BRDF data may come in one of two forms: as regularly spaced tabularized data, or as a large number of irregularly spaced individual samples. Regularly spaced data makes efficient lookups possible. (For example, if we have a 4D table of values for an anisotropic BRDF, indexed by $(\theta_i, \phi_i, \theta_o, \phi_o)$, then, given a pair of directions, we just need to index into the table and interpolate between the values.) However, it can be difficult to acquire BRDF values with such a regular spacing. Acquiring good measurements for directions close to the horizon can be particularly difficult, for example. Irregular sample spacings avoid these issues, though are not as easy to use for rendering. (Irregular sets of BRDF samples are often resampled in a preprocess to generate a set of regular samples.)

Therefore, `pbrt` has implementations of two `BxDF`s for using measured reflection data for rendering. The first, `IrregIsotropicBRDF` interpolates from irregularly spaced samples of isotropic BRDFs. The second, `RegularHalfangleBRDF`, supports the regular sampling used by Matusik et al.'s data set (2003a, 2003b). Both of these are used by the `MeasuredMaterial` defined in Section 9.2.4.

BxDF 428
IrregIsotropicBRDF 464
MeasuredMaterial 489
RegularHalfangleBRDF 467

8.6.1 IRREGULAR ISOTROPIC MEASURED BRDF

`IrregIsotropicBRDF` supports irregularly sampled isotropic BRDF data. Given a pair of directions, it finds a few BRDF samples that are nearby in the space of incident and outgoing directions and interpolates between their values, based on how close they are to the actual directions.

⟨*BxDF Declarations*⟩ +≡
```
class IrregIsotropicBRDF : public BxDF {
public:
    ⟨IrregIsotropicBRDF Public Methods 465⟩
private:
    ⟨IrregIsotropicBRDF Private Data 465⟩
};
```

Finding a good encoding for the BRDF directions is important. Consider a table of isotropic BRDF samples, where each sample records the BRDF value for a given pair of directions defined by $(\theta_i, \phi_i, \theta_o, \phi_o)$. This is a natural form for measured data to be stored in, but it is not a good one for resampling and interpolation. One shortcoming of this representation is that the isotropy of the BRDF isn't reflected in the representation. Recall that for isotropic BRDFs rotation about the normal direction leaves the value unchanged, such that

$$f_r((\theta_i, \phi_i), (\theta_o, \phi_o)) = f_r((\theta_i, \phi_i + \delta), (\theta_o, \phi_o + \delta))$$

for all δ values. Therefore, a better approach would be to store the samples indexed by the two θ angles and the difference between ϕ directions, $(\theta_i, \theta_o, \Delta\phi)$, where $\Delta\phi = \phi_i - \phi_o$, thus reflecting this structure in the data. This gives a mapping from directions to a three-tuple:

$$m(\theta_i, \phi_i, \theta_o, \phi_o) \rightarrow (\theta_i, \theta_o, \phi_i - \phi_o)$$

If we did not account for the isotropy with a mapping like this one, and just did a 4D table lookup, then the closest relevant BRDF samples often would not be found for a particular pair of directions.

Tabularizing the data using the two θ angles and the ϕ difference has problems of its own, however. One shortcoming of this representation is that reciprocity isn't reflected in the mapping: interchanging the two directions maps to a different three-tuple unless both directions happen to have equal θ values.

Another shortcoming is that distance between samples isn't well correlated to actual distance between directions. For example, consider the pair of (θ, ϕ) directions $(.01, 0)$ and $(.01, \pi)$ and the pair of directions $(\pi/2 - .01, 0)$ and $(\pi/2 - .01, \pi)$. Both have $\Delta\phi = \pi$, but the first pair of directions is almost pointing in the same direction (toward the top of the hemisphere), and the second pair is pointing in nearly opposite directions at the horizon. This mapping also doesn't lead to a meaningful distance between samples due to a discontinuity in $\Delta\phi$: given two pairs of directions with equal θ values but where $\Delta\phi = 0$ for the first and $\Delta\phi = 2\pi - .01$ for the second, the mapping doesn't reflect that the two directions are actually close together.

We'd therefore like to use a mapping function for isotropic BRDFs that takes a 4D pair of directions ω_i and ω_o and converts them into a point in a three-dimensional space such that the distance between two points is meaningful, the isotropy of the BRDF is reflected, and reciprocity is represented. For the `IrregIsotropicBRDF`, we'll implement a mapping proposed by Marschner, denoted by ϕ_3 in his thesis (Marschner 1998; Section 5.6.3):

$$m(\theta_i, \phi_i, \theta_o, \phi_o) \rightarrow \left(\sin\theta_i \sin\theta_o, \Delta\phi/\pi, \cos\theta_i \cos\theta_o\right),$$

BxDF 428
IrregIsotropicBRDF 464

where $\Delta\phi = (\phi_i - \phi_o)$, and $\Delta\phi$ is remapped to be in the range $[0, \pi]$. (Thanks to the isotropy assumption, $\Delta\phi$ values in $[\pi, 2\pi]$ are equivalent to $2\pi - \Delta\phi$, which is in $[0, \pi]$.)

This mapping accounts for isotropy and reciprocity, and the distance between two three tuples in the mapping's range has a meaningful relationship to the distance between their respective pairs of directions. The `BRDFRemap()` function implements this mapping.

⟨*BxDF Method Definitions*⟩ +≡
```
Point BRDFRemap(const Vector &wo, const Vector &wi) {
    float cosi = CosTheta(wi), coso = CosTheta(wo);
    float sini = SinTheta(wi), sino = SinTheta(wo);
    float phii = SphericalPhi(wi), phio = SphericalPhi(wo);
    float dphi = phii - phio;
    if (dphi < 0.) dphi += 2.f * M_PI;
    if (dphi > 2.f * M_PI) dphi -= 2.f * M_PI;
    if (dphi > M_PI) dphi = 2.f * M_PI - dphi;
    return Point(sini * sino, dphi / M_PI, cosi * coso);
}
```

Given this remapping function, we can define the structure that stores each BRDF sample. The three-tuple for the remapped directions is stored in `p` and the sample value is stored in `v`.

⟨*Reflection Declarations*⟩ ≡
```
struct IrregIsotropicBRDFSample {
    IrregIsotropicBRDFSample(const Point &pp, const Spectrum &vv)
        : p(pp), v(vv) { }
    Point p;
    Spectrum v;
};
```

When a `IrregIsotropicBRDF` is created by the `MeasuredMaterial::GetBSDF()` method, it is given a pointer to a three-dimensional kd-tree that stores the BRDF samples, where the sample positions are represented with the values returned by `BRDFRemap()`. Note that this kd-tree is managed by the `MeasuredMaterial` and is only used for lookups (i.e., in a read-only manner) by the code here.

⟨*IrregIsotropicBRDF Public Methods*⟩ ≡ 464
```
IrregIsotropicBRDF(const KdTree<IrregIsotropicBRDFSample> *d)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)), isoBRDFData(d) { }
```

⟨*IrregIsotropicBRDF Private Data*⟩ ≡ 464
```
const KdTree<IrregIsotropicBRDFSample> *isoBRDFData;
```

The `IrregIsotropicBRDF::f()` method first remaps the given pair of directions and then performs a series of kd-tree lookups, searching with a progressively wider search radius for nearby points until a few samples have been found. In general, it's preferable to smoothly interpolate between a few BRDF samples rather than just find the single closest sample for the lookup here.

BRDFRemap() 465
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BxDF 428
BxDFType 428
CosTheta() 426
IrregIsotropicBRDF 464
IrregIsotropicBRDFSample 465
KdTree 1029
MeasuredMaterial 489
MeasuredMaterial::
GetBSDF() 489
M_PI 1002
Point 63
SinTheta() 426
Spectrum 263
SphericalPhi() 292
Vector 57

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum IrregIsotropicBRDF::f(const Vector &wo,
                               const Vector &wi) const {
    Point m = BRDFRemap(wo, wi);
    float lastMaxDist2 = .001f;
    while (true) {
        ⟨Try to find enough BRDF samples around m within search radius 466⟩
    }
}
```

`IrregIsoProc` is the callback structure used by the `KdTree::Lookup()` method for each sample within the search radius. (The squared search radius is provided to the `Lookup()` method in `maxDist2`.) If not enough points are found, the squared search distance is doubled, and another lookup is performed.

⟨*Try to find enough BRDF samples around* `m` *within search radius*⟩ ≡ 466
```
IrregIsoProc proc;
float maxDist2 = lastMaxDist2;
isoBRDFData->Lookup(m, proc, maxDist2);
if (proc.nFound > 2 || lastMaxDist2 > 1.5f)
    return proc.v.Clamp() / proc.sumWeights;
lastMaxDist2 *= 2.f;
```

`IrregIsoProc` processes each BRDF sample within the search radius, accumulating a weighted sum of samples close to the lookup point.

⟨*BxDF Local Definitions*⟩ ≡
```
struct IrregIsoProc {
    ⟨IrregIsoProc Public Methods 466⟩
    Spectrum v;
    float sumWeights;
    int nFound;
};
```

⟨*IrregIsoProc Public Methods*⟩ ≡ 466
```
IrregIsoProc() { sumWeights = 0.f; nFound = 0; }
```

BRDF sample values are weighted with an *ad hoc* exponential falloff based on the squared distance between the lookup point and the sample point. The sum of these weights is accumulated so that the final value can be normalized.

⟨*IrregIsoProc Public Methods*⟩ +≡ 466
```
void operator()(const Point &p, const IrregIsotropicBRDFSample &sample,
                float d2, float &maxDist2) {
    float weight = expf(-100.f * d2);
    v += weight * sample.v;
    sumWeights += weight;
    ++nFound;
}
```

BRDFRemap() 465
IrregIsoProc 466
IrregIsoProc::nFound 466
IrregIsoProc::sumWeights 466
IrregIsoProc::v 466
IrregIsotropicBRDFSample 465
IrregIsotropicBRDFSample::v 465
KdTree::Lookup() 1032
Point 63
Spectrum 263
Spectrum::Clamp() 265
Vector 57

8.6.2 REGULAR HALFANGLE FORMAT

There are advantages to using BRDF samples with a regular spacing: looking up the value for a given direction is straightforward, not requiring a spatial data structure or other interpolation scheme, and it can be easier to implement Monte Carlo sampling methods based on regularly tabularized data. In this section we implement a `BxDF` that supports isotropic measured BRDFs stored in a format used by Matusik et al. (2003a). This format is well designed, and a large body of measured data is available for it.[6]

⟨*BxDF Declarations*⟩ +≡
```
class RegularHalfangleBRDF : public BxDF {
public:
    ⟨RegularHalfangleBRDF Public Methods 468⟩
private:
    ⟨RegularHalfangleBRDF Private Data 468⟩
};
```

As with the irregular BRDF data in Section 8.6.1, choosing a remapping of the usual (ω_i, ω_o) representation of pairs of directions is important for accurate representation of BRDFs with tabularized data. As was first observed by Rusinkiewicz (1998), the usual $(\theta_i, \phi_i, \theta_o, \phi_o)$ representation of directions has the shortcoming that important features in the BRDF can move in an irregular manner through its 4D space. Consider for example a surface with perfect specular reflection: as (θ_o, ϕ_o) vary, the specular reflection direction moves accordingly in (θ_i, ϕ_i). A different mapping of the pair of directions that instead ensured that this feature was aligned to one of the 4D coordinate axes would lead to more accurate lookup and interpolation of tabularized reflection data and would also be better suited to efficiently fitting data with basis functions.

Rusinkiewicz proposed a mapping based on the half-angle vector $\omega_h = \omega_i \widehat{+} \omega_o$ and the difference vector ω_d,

$$m(\theta_i, \phi_i, \theta_o, \phi_o) \rightarrow (\theta_h, \phi_h, \theta_d, \phi_d) \quad [8.10]$$

where the difference vector is found by finding the rotation that aligns ω_h with the (0, 0, 1) direction and then applying that rotation to ω_i. Figure 8.23 shows the effect of remapping the Torrance–Sparrow BRDF using Equation (8.10). Notice how the specular peak remains aligned along the $+z$ axis, even as the outgoing direction varies.

`RegularHalfangleBRDF` takes a pointer to a 3D table of sampled isotropic BRDF values, indexed by $(\sqrt{\theta}_h, \theta_d, \phi_d)$, where the number of samples in each direction is respectively `nThetaH`, `nThetaD`, and `nPhiD`. Due to the assumption of isotropy, the ϕ_h term can be dropped. The square root of θ_h is taken in order to slightly increase the sampling rate for θ_h values close to zero. The most rapid variation of sample values for many BRDFs is in this region of the domain, as specular highlights are generally well aligned to the ω_h axis after the remapping, so small changes in θ_h close to zero can represent significant changes in the function's value.

`BxDF` 428

`RegularHalfangleBRDF` 467

6 Matusik et al.'s measurement data for approximately 100 different materials is available from *www.merl.com/brdf*.

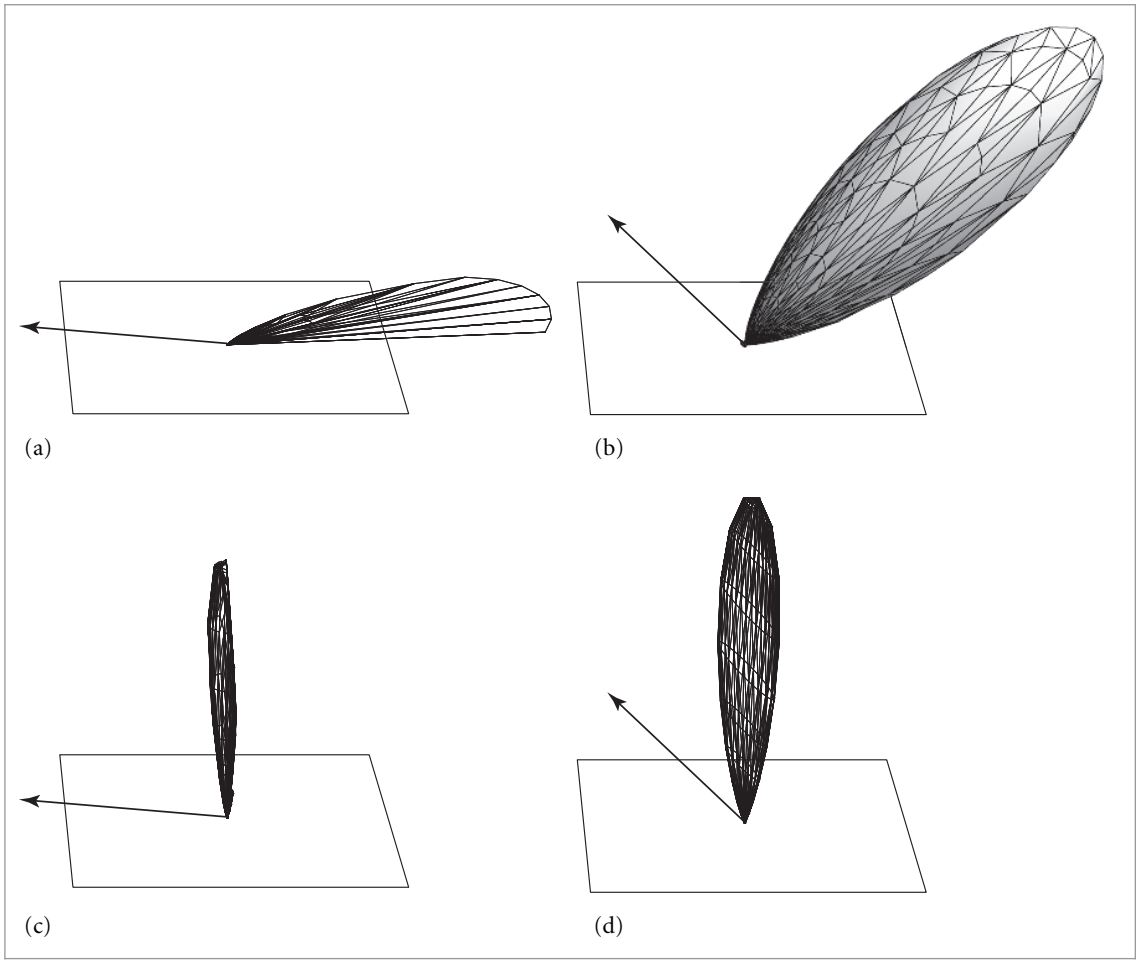


Figure 8.23: (a, b) The Torrance–Sparrow BRDF for two outgoing directions ω_o. (c, d) Torrance–Sparrow BRDF for the same two outgoing directions, after remapping the function with Equation (8.10). Note that the specular lobe remains aligned with the $+z$ axis even as the outgoing direction changes.

⟨*RegularHalfangleBRDF Public Methods*⟩ ≡ 467
```
RegularHalfangleBRDF(const float *d, uint32_t nth, uint32_t ntd,
                     uint32_t npd)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)), brdf(d),
      nThetaH(nth), nThetaD(ntd), nPhiD(npd) { }
```

⟨*RegularHalfangleBRDF Private Data*⟩ ≡ 467
```
const float *brdf;
const uint32_t nThetaH, nThetaD, nPhiD;
```

BSDF_GLOSSY 428
BSDF_REFLECTION 428
BxDF 428
BxDFType 428
RegularHalfangleBRDF 467

To compute the BRDF's value for a given pair of directions, it's necessary to apply the mapping given by Equation (8.10) to remap the directions to the half-angle coordi-

nate system, compute the corresponding indices, and to then perform the table lookup. `RegularHalfangleBRDF` returns the single closest sample value without further interpolation. Trilinear interpolation could be easily implemented, though the available data sets tend to have high sampling rates, making interpolation unnecessary.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum RegularHalfangleBRDF::f(const Vector &wo,
                                 const Vector &wi) const {
    ⟨Compute ωh and transform ωi to halfangle coordinate system 469⟩
    ⟨Compute index into measured BRDF tables 469⟩
    return Spectrum::FromRGB(&brdf[3*index]);
}
```

The half-angle direction ω_h is easily computed as usual. The rotation matrix that rotates ω_h to be aligned with (0, 0, 1) is then computed by finding the product of the matrices for rotating $-\phi_h$ about the z axis and then rotating $-\theta_h$ about the y axis.[7] The code in this fragment directly computes the entries of the product of these two matrices, the rows of which are represented here as `whx`, `why`, and `wh`, respectively. Multiplying ω_i by this matrix gives ω_d.

⟨*Compute ωh and transform ωi to halfangle coordinate system*⟩ ≡ 469
```
Vector wh = Normalize(wi + wo);
float whTheta = SphericalTheta(wh);
float whCosPhi = CosPhi(wh), whSinPhi = SinPhi(wh);
float whCosTheta = CosTheta(wh), whSinTheta = SinTheta(wh);
Vector whx(whCosPhi * whCosTheta, whSinPhi * whCosTheta, -whSinTheta);
Vector why(-whSinPhi, whCosPhi, 0);
Vector wd(Dot(wi, whx), Dot(wi, why), Dot(wi, wh));
```

Given the difference vector ω_d, we can directly compute the corresponding spherical coordinates. Due to the BRDF's reciprocity, its value for $\omega_d = (\theta_d, \phi_d)$ is equal to its value for $\omega_d = (\theta_d, \phi_d + \pi)$, so if `wDiffPhi` is greater than π, then π is subtracted from it. Integer indices for each value are computed by the fragment ⟨*Compute indices* `whThetaIndex, wdThetaIndex, wdPhiIndex`⟩ and then the index for a 3D array lookup is stored in `index`.

⟨*Compute* `index` *into measured BRDF tables*⟩ ≡ 469
```
float wdTheta = SphericalTheta(wd), wdPhi = SphericalPhi(wd);
if (wdPhi > M_PI) wdPhi -= M_PI;
⟨Compute indices whThetaIndex, wdThetaIndex, wdPhiIndex 470⟩
int index = wdPhiIndex + nPhiD * (wdThetaIndex + whThetaIndex * nThetaD);
```

Given the three angles, their respective indices into the BRDF data table are found using the `REMAP()` macro, which maps the given floating-point value `V`, expected to be between 0 and `MAX` to an integer from 0 to `COUNT-1`.

M_PI 1002
RegularHalfangleBRDF 467
RegularHalfangleBRDF::brdf 468
RegularHalfangleBRDF::nPhiD 468
RegularHalfangleBRDF::nThetaD 468
Spectrum 263
Spectrum::FromRGB() 277
Vector 57

7 This matrix could be computed more efficiently, without using the trigonometric functions used here, with the technique described by Akenine-Möller and Hughes (1999). However, the value of θ_h is also needed below for the table lookup, so here we just compute the matrix using the trigonometric approach.

⟨*Compute indices* whThetaIndex, wdThetaIndex, wdPhiIndex⟩ ≡ 469

```
#define REMAP(V, MAX, COUNT) \
    Clamp(int((V) / (MAX) * (COUNT)), 0, (COUNT)-1)
int whThetaIndex = REMAP(sqrtf(max(0.f, whTheta / (M_PI / 2.f))),
                         1.f, nThetaH);
int wdThetaIndex = REMAP(wdTheta, M_PI / 2.f, nThetaD);
int wdPhiIndex = REMAP(wdPhi, M_PI, nPhiD);
#undef REMAP
```

FURTHER READING

Phong (1975) developed an early empirical reflection model for glossy surfaces in computer graphics. Although not reciprocal or energy conserving, it was a cornerstone of the first synthetic images of non-Lambertian objects. The Torrance–Sparrow microfacet model is described in Torrance and Sparrow (1967); it was first introduced to graphics by Blinn (1977), and a variant of it was used by Cook and Torrance (1981, 1982). The Oren–Nayar Lambertian model is described in their 1994 SIGGRAPH paper (Oren and Nayar 1994).

Hall's book collected and described the state of the art in physically based surface reflection models for graphics in 1989; it remains a seminal reference (Hall 1989). It discusses the physics of surface reflection in detail, with many pointers to the original literature and with many tables of useful measured data about reflection from real surfaces.

Moravec (1981) was the first to apply a wave optics model to graphics. This area has also been investigated by Bahar and Chakrabarti (1987). Beckmann and Spizzichino (1963) developed an early physical optics model of surface reflection, which was used by Kajiya (1985) to derive an anisotropic reflection model for computer graphics. Beckmann and Spizzichino's work was built upon by He et al. (1991), who developed a sophisticated reflection model that modeled a variety of types of surface reflection, and Stam (1999), who applied wave optics to model diffraction effects.

Nayar, Ikeuchi, and Kanade (1991) have shown that some reflection models based on physical (wave) optics have substantially similar characteristics to those based on geometric optics. The geometric optics approximations don't seem to cause too much error in practice, except on very smooth surfaces. This is a helpful result, giving experimental basis to the general belief that wave optics models aren't usually worth their computational expense for computer graphics applications.

The effect of the polarization of light is ignored in pbrt, although for some scenes it can be an important effect; see, for example, the paper by Tannenbaum, Tannenbaum, and Wozny (1994) for information about how to extend a renderer to account for this effect. Similarly, the fact that indices of refraction of real-world objects usually vary as a function of wavelength is also not modeled here; see both Section 11.8 of Glassner's book (1995) and Devlin et al.'s survey article for information about these issues and references to previous work (Devlin et al. 2002). Fluorescence, where light is reflected at different wavelengths than the incident illumination, is also not modeled by pbrt; see Glassner (1994) and Wilkie et al. (2006) for more information on this topic.

M_PI 1002
RegularHalfangleBRDF::
nPhiD 468
RegularHalfangleBRDF::
nThetaD 468
RegularHalfangleBRDF::
nThetaH 468

Notable anisotropic BRDF models for computer graphics include those of Poulin and Fournier (1990), Ward (1992), and Schlick (1993). Schlick's model is both computationally efficient and easy to use with importance sampling for Monte Carlo integration. Other good references for anisotropic models are papers by Lu, Koenderink, and Kappers (1999) and Ashikhmin and Shirley (2000, 2002).

Ashikhmin, Premoze, and Shirley (2000) developed techniques for computing self-shadowing terms for arbitrary microfacet distributions, without requiring the assumptions that Torrance and Sparrow did. Their solutions cannot be evaluated in closed form, but must be approximated numerically. Weyrich et al. (2009) have developed methods to infer a microfacet distribution that matches a measured or desired reflection distribution. Remarkably, they show that physical surfaces that match the desired reflection distribution can be manufactured, with a good match to the desired distribution.

Stam (2001) also derived a generalization of the Cook–Torrance model for transmission, and more recently Walter et al. (2007) revisited this problem, addressing issues of finding appropriate microfacet and self-shadowing functions for transmission and finding effective importance sampling approaches for Monte Carlo integration. There are a number of important subtleties that are not accounted for in the simple approach taken in the `BRDFToBTDF` class in this chapter.

Improvements in data-acquisition technology have led to increasing amounts of measured real-world BRDF data, even including BRDFs that are spatially varying (sometimes called "bidirectional texture functions," BTFs). Matusik et al. (2003a, 2003b) assembled a large database of measured isotropic BRDF data; the `RegularHalfangleBRDF` uses data stored in their representation. See Müller et al. (2005) for a survey of recent work in BRDF measurement. Sun et al. (2007) measured BRDFs as they change over time—for example, due to paint drying, a wet surface becoming dry, or dust accumulating.

Fitting measured BRDF data to parametric reflection models is a difficult problem. Rusinkiewicz (1998) made the influential observation that reparameterizing the measured data can make it substantially easier to compress or fit to models; this topic has been further investigated by Stark et al. (2005). Ngan et al. (2005) analyzed the effectiveness of a variety of BRDF models for fitting measured data and showed that models based on the half-angle vector, rather than a reflection vector, tended to be more effective. See also the paper on this topic by Edwards et al. (2005).

Our approach for inteprolating irregular BRDF measurements in Section 8.6 is *ad hoc*. See Zickler et al. (2005) for a better-grounded method based on using radial basis functions (RBFs)—they use them to interpolate irregularly sampled 5D spatially varying BRDFs. More recently, Weistroffer et al. (2007) have shown how to efficiently represent scattered reflectance data with RBFs without needing to resample it to have regular spacing and Wang et al. (2008) demonstrated a successful approach for acquiring spatially varying anisotropic BRDFs.

Kajiya and Kay (1989) developed a reflection model for hair based on a model of individual hairs as cylinders with diffuse and glossy reflection properties. Their model determines the overall reflection from these cylinders, accounting for the effect of variation in surface normal over the hemisphere along the cylinder. See also the paper by Banks (1994), which discusses shading models for 1D primitives like hair. More recently,

BRDFToBTDF 431
RegularHalfangleBRDF 467

Goldman (1997) developed a probabilistic shading model that models reflection from collections of short hairs, and Marschner et al. (2003) developed a more accurate model of light scattering from long hair. A number of other specific types of surfaces have received specific attention from researchers, leading to specialized reflection models being developed for them, including Marschner et al.'s work on rendering wood (Marschner et al. 2005), Sattler et al.'s approach for rendering cloth (Sattler et al. 2003), Irawan's work on cloth rendering (Irawan 2008), and Günther et al.'s investigation of car paint (Günther et al. 2005).

A number of researchers have investigated BRDFs based on modeling the small-scale geometric features of a reflective surface. This work includes the computation of BRDFs from bump maps by Cabral, Max, and Springmeyer (1987), Fournier's normal distribution functions (Fournier 1992), and Westin, Arvo, and Torrance (1992), who applied Monte Carlo ray tracing to statistically model reflection from microgeometry and represented the resulting BRDFs with spherical harmonics.

EXERCISES

❶ 8.1 A consequence of Fermat's principle from optics is that light traveling from a point p_1 in a medium with index of refraction η_1 to a point p_2 in a medium with index of refraction η_2 will follow a path that minimizes the time to get from the first point to the second point. Snell's law can be shown to follow from this fact directly.

Consider light traveling between two points p_1 and p_2 separated by a planar boundary. The light could potentially pass through the boundary while traveling from p_1 to p_2 at any point on the boundary (see Figure 8.24, which shows two such possible points p' and p''). Recall that the time it takes light to travel between two points in a medium with a constant index of refraction is proportional to the distance between them times the index of refraction in the medium. Using this fact, show that the point p' on the boundary that minimizes the total time to travel from p_1 to p_2 is the point where $\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$.

❸ 8.2 Read the papers of Wolff and Kurlander (1990) and Tannenbaum, Tannenbaum, and Wozny (1994) and apply some of the techniques described to modify `pbrt` to model the effect of light polarization. Set up scenes and render images of them that demonstrate a significant difference when polarization is accurately modeled.

❸ 8.3 Construct a scene with an actual geometric model of a rough plane with a large number of mirrored microfacets and illuminate with an area light source.[8] Place the camera in the scene such that a very large number of microfacets are in each pixel's area, and render images of this scene using hundreds or thousands of pixel samples. Compare the result to using a flat surface with a

8 An area light and not a point or directional light is necessary due to subtleties in how lights are seen in specular surfaces. With the light transport algorithms used in `pbrt`, infinitesimal point sources are never visible in mirrored surfaces. This is a typical limitation of ray-tracing renderers and usually not bothersome in practice.

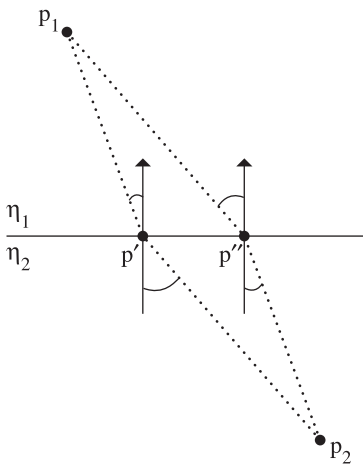


Figure 8.24: Derivation of Snell's Law. Snell's law can be derived using Fermat's principle, which says that light will follow the path that takes the least amount of time to pass between two points. The angle of refraction θ at the boundary between two media can thus be shown to be the one that minimizes the time spent going from p_1 to a point p on the boundary plus the time spent traveling the distance from that point to p_2.

microfacet-based BRDF model. How well can you get the two approaches to match if you try to tune the microfacet BRDF parameters? Can you construct examples where images rendered with the true microfacets are actually visibly more realistic due to better modeling the effects of masking, self-shadowing, and interreflection between microfacets?

❷ 8.4 Implement the approach for rendering transmission through microfacet-based surfaces described by Walter et al. (2007), including their self-shadowing term and importance sampling techniques. Compare images rendered with this approach to images rendered using the `BRDFToBTDF` adapter class defined in this chapter; does the more sound physical basis for their method lead to visually different images?

❸ 8.5 Replace the implementation of the `IrregIsotropicBRDF` with a more soundly grounded scattered BRDF interpolation technique such as the one described by Zickler et al. (2005) or Weistroffer et al. (2007). What sort of visual differences are evident with images rendered with your implementation compared to the current implementation in `pbrt`? How does the execution time change when your implementation is used for rendering images with irregularly sampled measured BRDF data?

❸ 8.6 Extend `pbrt` to be able to more accurately render interesting surfaces like wood (Marschner et al. 2005), cloth (Sattler et al. 2003), or car paint (Günther et al. 2005). Render images showing better visual results than when existing reflection functions in `pbrt` are used for these effects.

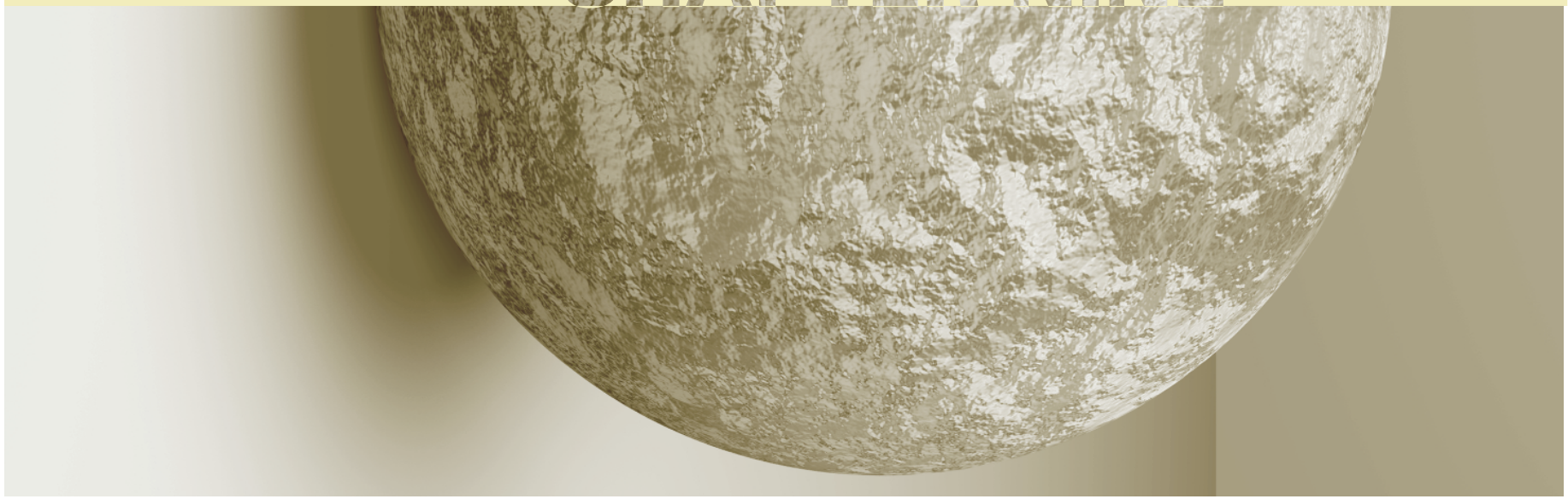
BRDFToBTDF 431
IrregIsotropicBRDF 464

3 8.7 Implement a simulation-based approach to modeling reflection from complex microsurfaces, such as the one described by Westin, Arvo, and Torrance (1992). Modify `pbrt` so that you can provide a description of the microgeometry of a complex surface (like cloth, velvet, etc.), fire rays at the geometry from a variety of incident directions, and record the distribution and throughput for the rays that leave the surface. (You will likely need to modify the `PathIntegrator` from Chapter 15 to determine the distribution of outgoing light.) Record the distribution in a three-dimensional table if the surface is isotropic or a four-dimensional table if it is anisotropic, and use the table to compute BRDF values for rendering images. Demonstrate interesting reflection effects from complex surfaces using this approach. Investigate how the size of the table and the number of samples taken to compute entries in the table affect the accuracy of the final result.

PathIntegrator 766

This page intentionally left blank

CHAPTER NINE





MATERIALS

The low-level BRDFs and BTDFs introduced in the previous chapter address only part of the problem of describing how a surface scatters light. Although they describe how light is scattered at a particular point on a surface, the renderer needs to determine *which* BRDFs and BTDFs to use at that point and how to set their parameters. In this chapter, we describe a procedural shading mechanism that determines the BRDFs and BTDFs to use at points on surfaces.

The basic idea is that a *surface shader* is bound to each primitive in the scene. The surface shader is represented by an instance of the `Material` interface class, which has a method that takes a point to be shaded and returns a `BSDF` object. The `BSDF` class holds a set of `BxDF`s that collectively describe scattering at a point. `Material`s, in turn, use instances of the `Texture` class (to be defined in the next chapter) to determine the material properties at particular points on surfaces. For example, an `ImageTexture` might be used to modulate the color of diffuse reflection across a surface. This is a somewhat different shading paradigm than many rendering systems use; it is common practice to combine the function of the surface shader and the lighting integrator (see Chapter 15) into a single module and have the shader return the color of reflected light at the point. However, by separating these two components and having the `Material` return a `BSDF`, `pbrt` is better able to handle a variety of light transport algorithms.

9.1 BSDFs

The `BSDF` class represents a collection of BRDFs and BTDFs. Grouping them in this manner allows the rest of the system to work with composite `BSDF`s directly, rather than having to consider all of the components they may have been built from. Equally important, the `BSDF` class hides some of the details of shading normals from the rest of the system. Shading normals, either from per-vertex normals in triangle meshes or from bump mapping, can substantially improve the visual richness of rendered scenes, but because they are

an *ad hoc* construct, they are tricky to incorporate into a physically based renderer. The issues that they introduce are handled in the BSDF implementation.

⟨*BSDF Declarations*⟩ +≡
```
class BSDF {
public:
    ⟨BSDF Public Methods 479⟩
    ⟨BSDF Public Data 479⟩
private:
    ⟨BSDF Private Methods 483⟩
    ⟨BSDF Private Data 479⟩
};
```

The BSDF constructor takes a DifferentialGeometry object that represents the shading differential geometry, the true surface normal ngeom, and, optionally, the index of refraction of the medium enclosed by the surface. It stores these values in member variables and constructs an orthonormal coordinate system with the shading normal as one of the axes; this will be useful for transforming directions to and from the BxDF coordinate system described in Section 8.1. Throughout this section, we will use the convention that $\mathbf{n}_s$ denotes the shading normal and $\mathbf{n}_g$ the geometric normal (Figure 9.1).

⟨*BSDF Method Definitions*⟩ ≡
```
BSDF::BSDF(const DifferentialGeometry &dg, const Normal &ngeom,
           float e)
    : dgShading(dg), eta(e) {
    ng = ngeom;
```

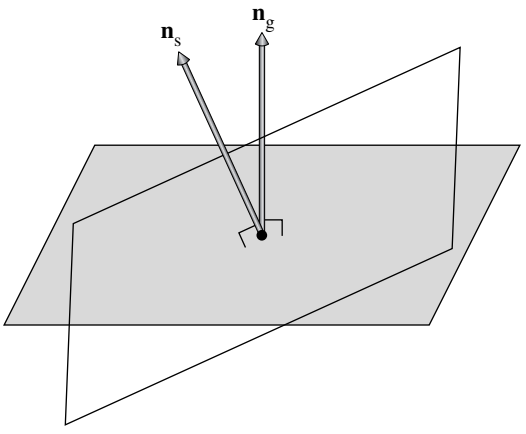


Figure 9.1: The geometric normal, $\mathbf{n}_g$, defined by the surface geometry, and the shading normal, $\mathbf{n}_s$, given by per-vertex normals and/or bump mapping, will generally define different hemispheres for integrating incident illumination to compute surface reflection. This inconsistency is important to handle carefully since it can otherwise lead to artifacts in images.

BSDF 478
BSDF::nBxDFs 479
BxDF 428
Cross() 62
DifferentialGeometry 102
Normal 65
Vector::Normalize() 63

```
    nn = dgShading.nn;
    sn = Normalize(dgShading.dpdu);
    tn = Cross(nn, sn);
    nBxDFs = 0;
}
```

⟨*BSDF Public Data*⟩ ≡ 478
```
const DifferentialGeometry dgShading;
const float eta;
```

⟨*BSDF Private Data*⟩ ≡ 478
```
Normal nn, ng;
Vector sn, tn;
```

The BSDF implementation stores only a fixed limited number of individual BxDF components. It could easily be extended to allocate more space if more components were given to it, although this isn't necessary for any of the Material implementations in pbrt thus far, and the current limit of eight is plenty for almost all practical applications.

⟨*BSDF Inline Method Definitions*⟩ ≡
```
inline void BSDF::Add(BxDF *b) {
    Assert(nBxDFs < MAX_BxDFS);
    bxdfs[nBxDFs++] = b;
}
```

⟨*BSDF Private Data*⟩ +≡ 478
```
int nBxDFs;
#define MAX_BxDFS 8
BxDF *bxdfs[MAX_BxDFS];
```

For other parts of the system that need additional information about the particular BRDFs and BTDFs present, two different methods return the number of BxDFs stored by the BSDF; the second only returns the number that matches a particular set of BxDFType flags. Note that we could use a single function with a default parameter value of BSDF_ALL, but the no-argument variant of BSDF::NumComponents() is called frequently, so we would like to avoid the overhead of checking the flags of the BxDFs individually in that case.

⟨*BSDF Public Methods*⟩ ≡ 478
```
int NumComponents() const { return nBxDFs; }
int NumComponents(BxDFType flags) const;
```

The BSDF also has methods that perform transformations to and from the local coordinate system expected by BxDFs. Recall that, in this coordinate system, the surface normal is along the z axis (0, 0, 1), the primary tangent is (1, 0, 0), and the secondary tangent is (0, 1, 0). The transformation of directions into "shading space" simplifies many of the BxDF implementations in Chapter 8. Given three orthonormal vectors $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{n}$ in world space, the matrix $\mathbf{M}$ that transforms vectors in world space to the local reflection space is

$$\mathbf{M} = \begin{pmatrix} \mathbf{s}_x & \mathbf{s}_y & \mathbf{s}_z \\ \mathbf{t}_x & \mathbf{t}_y & \mathbf{t}_z \\ \mathbf{n}_x & \mathbf{n}_y & \mathbf{n}_z \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{t} \\ \mathbf{n} \end{pmatrix}.$$

Assert() 1005
BSDF 478
BSDF::bxdfs 479
BSDF::nBxDFs 479
BSDF::NumComponents() 479
BSDF_ALL 428
BxDF 428
BxDFType 428
DifferentialGeometry 102
Material 483
Normal 65
Vector 57

To confirm this yourself, consider, for example, the value of $\mathbf{M}$ times the surface normal $\mathbf{n}$, $\mathbf{Mn} = (\mathbf{s} \cdot \mathbf{n}, \mathbf{t} \cdot \mathbf{n}, \mathbf{n} \cdot \mathbf{n})$. Since $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{n}$ are all orthonormal, the x and y components of $\mathbf{Mn}$ are zero. Since $\mathbf{n}$ is normalized, $\mathbf{n} \cdot \mathbf{n} = 1$. Thus, $\mathbf{Mn} = (0, 0, 1)$, as expected.

In this case, we don't need to compute the inverse transpose of $\mathbf{M}$ to transform normals (recall the discussion of transforming normals in Section 2.8.3). Because $\mathbf{M}$ is an orthogonal matrix (its rows and columns are mutually orthogonal), its inverse is equal to its transpose, so it is its own inverse transpose already.

⟨*BSDF Public Methods*⟩ +≡ 478

```
Vector WorldToLocal(const Vector &v) const {
    return Vector(Dot(v, sn), Dot(v, tn), Dot(v, nn));
}
```

The method that takes vectors back from local space to world space uses the transpose to invert $\mathbf{M}$ before doing the appropriate dot products.

⟨*BSDF Public Methods*⟩ +≡ 478

```
Vector LocalToWorld(const Vector &v) const {
    return Vector(sn.x * v.x + tn.x * v.y + nn.x * v.z,
                  sn.y * v.x + tn.y * v.y + nn.y * v.z,
                  sn.z * v.x + tn.z * v.y + nn.z * v.z);
}
```

Shading normals can cause a variety of undesirable artifacts in practice (Figure 9.2). Figure 9.2(a) shows a *light leak*: the geometric normal indicates that ω_i and ω_o lie on opposite sides of the surface, so if the surface is not transmissive, the light should have no contribution. However, if we directly evaluate the scattering equation, Equation (5.8), about the hemisphere centered around the shading normal, we will incorrectly incorporate the light from ω_i. This case demonstrates that $\mathbf{n}_s$ can't just be used as a direct replacement for $\mathbf{n}_g$ in rendering computations.

Figure 9.2(b) shows a similar tricky situation: the shading normal indicates that no light should be reflected to the viewer, since it is not in the same hemisphere as the illumination, while the geometric normal indicates that they are in the same hemisphere. Direct use of $\mathbf{n}_s$ would cause ugly black spots on the surface where this situation happens.

Fortunately, there is an elegant solution to these problems. When evaluating the BSDF, we use the geometric normal to decide if we should be evaluating reflection or transmission: if ω_i and ω_o lie in the same hemisphere with respect to $\mathbf{n}_g$, we evaluate the BRDFs, and otherwise we evaluate the BTDFs. In evaluating the scattering equation, however, the dot product of the normal and the incident direction is still taken with the shading normal, rather than the geometric normal.

BSDF::nn 479
BSDF::sn 479
BSDF::tn 479
BxDF 428
Dot() 60
Vector 57

Now it should be clear why `pbrt` requires `BxDF`s to evaluate their values without regard to whether ω_i and ω_o are in the same or different hemispheres. Thus, light leaks are avoided, since we will only evaluate the BTDFs for the situation in Figure 9.2(a), giving no reflection for a purely reflective surface. Similarly, black spots are avoided since we will evaluate the BRDFs for the situation in Figure 9.2(b), even though the shading normal would suggest that the directions are in different hemispheres.

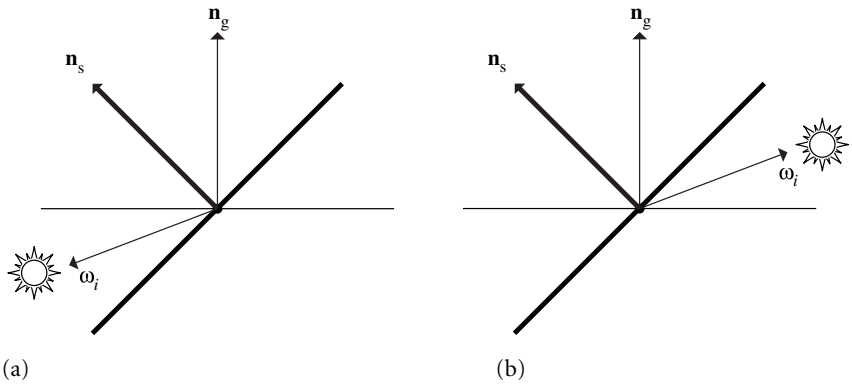


Figure 9.2: The Two Types of Errors That Result from Using Shading Normals. (a) A light leak: the geometric normal indicates that the light is on the back side of the surface, but the shading normal indicates the light is visible (assuming a reflective and not transmissive surface). (b) A dark spot: the geometric normal indicates that the surface is illuminated, but the shading normal indicates that the viewer is behind the lit side of the surface.

Given these conventions, the method that evaluates the BSDF for a given pair of directions is straightforward. It starts by transforming the world space direction vectors to local BSDF space and then determines whether it should use the BRDFs or the BTDFs. It then loops over the appropriate set and evaluates the sum of their contributions.

⟨*BSDF Method Definitions*⟩ +≡

```
Spectrum BSDF::f(const Vector &woW, const Vector &wiW,
                 BxDFType flags) const {
    Vector wi = WorldToLocal(wiW), wo = WorldToLocal(woW);
    if (Dot(wiW, ng) * Dot(woW, ng) > 0) // ignore BTDFs
        flags = BxDFType(flags & ~BSDF_TRANSMISSION);
    else // ignore BRDFs
        flags = BxDFType(flags & ~BSDF_REFLECTION);
    Spectrum f = 0.;
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i]->MatchesFlags(flags))
            f += bxdfs[i]->f(wo, wi);
    return f;
}
```

BSDF 478
BSDF::bxdfs 479
BSDF::nBxDFs 479
BSDF::WorldToLocal() 480
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDF 428
BxDF::f() 429
BxDF::MatchesFlags() 429
BxDF::rho() 430
BxDFType 428
Dot() 60
RNG 1003
Spectrum 263
Vector 57

pbrt also provides BSDF methods that return the summed reflectance values of their individual BxDFs. These methods just loop over the BxDFs and call the appropriate BxDF::rho() methods and therefore won't be shown here. A RNG must be passed to these methods so that they can generate random samples for BxDF implementations for use in Monte Carlo sampling algorithms if needed—recall the BxDF::rho() interface defined in Section 8.1.1.

⟨*BSDF Public Methods*⟩ +≡ 478
```
Spectrum rho(RNG &rng, BxDFType flags = BSDF_ALL,
             int sqrtSamples = 6) const;
Spectrum rho(const Vector &wo, RNG &rng, BxDFType flags = BSDF_ALL,
             int sqrtSamples = 6) const;
```

9.1.1 BSDF MEMORY MANAGEMENT

For each camera ray that intersects geometry in the scene, one or more BSDF objects will be created by the SurfaceIntegrator in the process of computing the reflected radiance from the intersection point. (Integrators that account for multiple interreflections of light will generally create a number of BSDFs along the way.) Each of these BSDFs in turn has a number of BxDFs stored inside it, as returned by the Materials at the intersection points. A naïve implementation of this process would use new and delete to dynamically allocate storage for both the BSDF as well as each of the BxDFs that it holds.

Unfortunately, such an approach is inefficient—too much time is spent in the dynamic memory management routines for a series of small memory allocations. Instead, the implementation here uses a specialized allocation scheme based on the MemoryArena class described in Section A.5.4 in Appendix A. MemoryArena allocates a large block of memory and responds to allocation requests via the MemoryArena::Alloc() call by returning successive sections of that block. It does not support freeing individual allocations, but instead frees all of them simultaneously when the MemoryArena::FreeAll() method is called. The result of this approach is that both allocation and freeing of memory are extremely efficient.

For the convenience of code that allocates BSDFs and BxDFs (e.g., the Materials in this chapter), there is a macro that hides some of the messiness of using the memory arena approach. Instead of using the new operator to allocate those objects like this:

```
BSDF *b = new BSDF;
BxDF *lam = new Lambertian(Spectrum(1.0));
```

code should instead be written with the BSDF_ALLOC() macro, like this:

```
BSDF *b = BSDF_ALLOC(arena, BSDF);
BxDF *lam = BSDF_ALLOC(arena, Lambertian)(Spectrum(1.0));
```

where arena is a MemoryArena. A MemoryArena object is generally passed down to the methods that need to allocate memory for BSDFs. For example, the SamplerRendererTask::Run() method allocates a MemoryArena for each rendering task and passes it to the integrators.

This macro calls the MemoryArena::Alloc() routine to allocate the appropriate amount of memory for the object and then uses the placement operator new to run the constructor for the object at the returned memory location.

⟨*BSDF Declarations*⟩ +≡
```
#define BSDF_ALLOC(arena, Type) new (arena.Alloc(sizeof(Type))) Type
```

The BSDF destructor is a private method, in order to ensure that it isn't inadvertently called, for example, due to an attempt to delete a BSDF. Making the destructor private

BSDF 478
BSDF_ALL 428
BSDF_ALLOC() 482
BxDF 428
BxDFType 428
Integrator 740
Material 483
MemoryArena 1015
MemoryArena::Alloc() 1016
MemoryArena::FreeAll() 1017
RNG 1003
SamplerRendererTask::Run() 30
Spectrum 263
SurfaceIntegrator 740
Vector 57

ensures a compile time error if it is called. Trying to `delete` memory allocated by the `MemoryArena` could lead to errors or crashes, since a pointer to the middle of memory managed by the `MemoryArena` would be passed to the system's dynamic memory freeing routine. Thus, a further implication of the allocation scheme here is that `BSDF` and `BxDF` destructors are never executed. This isn't a problem for the ones currently implemented in the system.

⟨*BSDF Private Methods*⟩ ≡ 478
```
~BSDF() { }
```

9.2 MATERIAL INTERFACE AND IMPLEMENTATIONS

The abstract `Material` class defines two methods that material implementations must provide, `Material::GetBSDF()` and `Material::GetBSSRDF()`. Implementations of these methods are responsible for determining the reflective properties at the given point on the surface and returning an instance of the `BSDF` class that describes them. The `Material` class is defined in the files `core/material.h` and `core/material.cpp`.

⟨*Material Declarations*⟩ ≡
```
class Material : public ReferenceCounted {
public:
    ⟨Material Interface 483⟩
};
```

The `Material::GetBSDF()` method is given two differential geometry objects. The first, `dgGeom`, represents the actual differential geometry at the ray intersection point, and the second, `dgShading`, represents possibly perturbed shading geometry, such as that from per-vertex normals in a triangle mesh. The material implementation may further perturb the shading geometry with bump mapping in the `GetBSDF()` method; the returned `BSDF` holds information about the final shading geometry at the point as well as the BRDF and BTDF components for the point.

⟨*Material Interface*⟩ ≡ 483
```
virtual BSDF *GetBSDF(const DifferentialGeometry &dgGeom,
                      const DifferentialGeometry &dgShading,
                      MemoryArena &arena) const = 0;
```

For materials that represent translucent materials that exhibit subsurface scattering, the `GetBSSRDF()` method returns a `BSSRDF` object that represents the object's subsurface scattering properties. The `BSSRDF` is defined later, in Section 11.6, after the foundations of volumetric scattering have been introduced. Because most materials don't have meaningful amounts of subsurface scattering, this method has a default implementation that just returns `NULL`.

BSDF 478
BSSRDF 598
DifferentialGeometry 102
Material 483
Material::GetBSDF() 483
Material::GetBSSRDF() 484
MemoryArena 1015
ReferenceCounted 1010

⟨*Material Interface*⟩ +≡ 483
```
virtual BSSRDF *GetBSSRDF(const DifferentialGeometry &dgGeom,
                          const DifferentialGeometry &dgShading,
                          MemoryArena &arena) const {
    return NULL;
}
```

Since the usual interface to the hit point used by `Integrators` is through an instance of the `Intersection` class, we will add two convenience methods to `Intersection`, `GetBSDF()` and `GetBSSRDF()`, that respectively return the `BSDF` or `BSSRDF` at the hit point. These methods call the `Differentialgeometry::ComputeDifferentials()` method to compute information about the projected size of the surface area around the intersection on the image plane for use in texture antialiasing and then forward the request to the `Primitive`, which in turn will call the corresponding `GetBSDF()` or `GetBSSRDF()` method of its `Material`. (See, for example, the `GeometricPrimitive::GetBSDF()` implementation.)

⟨*Intersection Method Definitions*⟩ ≡
```
BSDF *Intersection::GetBSDF(const RayDifferential &ray,
                            MemoryArena &arena) const {
    dg.ComputeDifferentials(ray);
    BSDF *bsdf = primitive->GetBSDF(dg, ObjectToWorld, arena);
    return bsdf;
}
```

The implementation of the `GetBSSRDF()` method is essentially the same, other than the fact that it calls the `Primitive::GetBSSRDF()` method.

⟨*Intersection Public Methods*⟩ ≡ 186
```
BSSRDF *GetBSSRDF(const RayDifferential &ray, MemoryArena &arena) const;
```

9.2.1 MATTEMATERIAL

The `MatteMaterial` material is defined in `materials/matte.h` and `materials/matte.cpp`. It is the simplest material in `pbrt` and describes a purely diffuse surface. It is parameterized by a spectral diffuse reflection value, `Kd`, and a scalar roughness value, `sigma`. If `sigma` has the value zero at the point on a surface, `MatteMaterial` returns a `Lambertian` BRDF; otherwise, the `OrenNayar` model is used. Like all of the other `Material` implementations in this chapter, it also takes an optional scalar texture that defines an offset function over the surface. If non-`NULL`, this texture is used to compute a shading normal at each point based on the function it defines. Figure 8.14 in the previous chapter shows the `MatteMaterial` material with the Killeroo model.

⟨*MatteMaterial Declarations*⟩ ≡
```
class MatteMaterial : public Material {
public:
    ⟨MatteMaterial Public Methods 485⟩
private:
    ⟨MatteMaterial Private Data 485⟩
};
```

BSDF 478
BSSRDF 598
DifferentialGeometry 102
DifferentialGeometry::ComputeDifferentials() 505
GeometricPrimitive::GetBSDF() 189
Integrator 740
Intersection 186
Intersection::dg 186
Intersection::ObjectToWorld 186
Intersection::primitive 186
Lambertian 447
Material 483
MatteMaterial 484
MemoryArena 1015
OrenNayar 449
Primitive 185
Primitive::GetBSDF() 187
Primitive::GetBSSRDF() 187
RayDifferential 69

⟨*MatteMaterial Public Methods*⟩ ≡ 484
```
MatteMaterial(Reference<Texture<Spectrum> > kd,
              Reference<Texture<float> > sig,
              Reference<Texture<float> > bump)
    : Kd(kd), sigma(sig), bumpMap(bump) {
}
```

⟨*MatteMaterial Private Data*⟩ ≡ 484
```
Reference<Texture<Spectrum> > Kd;
Reference<Texture<float> > sigma, bumpMap;
```

The `GetBSDF()` method puts the pieces together, determining the bump map's effect on the shading geometry, evaluating the textures, and allocating and returning the `BSDF`.

⟨*MatteMaterial Method Definitions*⟩ ≡
```
BSDF *MatteMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                             const DifferentialGeometry &dgShading,
                             MemoryArena &arena) const {
    ⟨Allocate BSDF, possibly doing bump mapping with bumpMap 485⟩
    ⟨Evaluate textures for MatteMaterial material and allocate BRDF 486⟩
    return bsdf;
}
```

If a bump map was provided to the `MatteMaterial` constructor, the `Material::Bump()` method is called to calculate the shading normal at the point. This method will be defined in the next section.

⟨*Allocate* `BSDF`, *possibly doing bump mapping with* `bumpMap`⟩ ≡ 485, 487, 489
```
DifferentialGeometry dgs;
if (bumpMap)
    Bump(bumpMap, dgGeom, dgShading, &dgs);
else
    dgs = dgShading;
BSDF *bsdf = BSDF_ALLOC(arena, BSDF)(dgs, dgGeom.nn);
```

Next, the `Textures` that give the values of the diffuse reflection coefficient and the roughness are evaluated; these may return constant values, look up values from image maps, or do complex procedural shading calculations to compute these values (the texture evaluation process is the subject of Chapter 10). Given these values, all that needs to be done is to allocate a `BSDF` and the appropriate BRDF component using the `BSDF` memory allocation macros and return the result. Because `Textures` may return negative values or values otherwise outside of the expected range, these values are clamped before they are passed to the BRDF constructor.

BSDF 478
DifferentialGeometry 102
Material::Bump() 495
MatteMaterial 484
MatteMaterial::bumpMap 485
MatteMaterial::Kd 485
MatteMaterial::sigma 485
MemoryArena 1015
Reference 1011
Spectrum 263
Texture 519

⟨*Evaluate textures for* `MatteMaterial` *material and allocate BRDF*⟩ ≡ 485

```
Spectrum r = Kd->Evaluate(dgs).Clamp();
float sig = Clamp(sigma->Evaluate(dgs), 0.f, 90.f);
if (sig == 0.)
    bsdf->Add(BSDF_ALLOC(arena, Lambertian)(r));
else
    bsdf->Add(BSDF_ALLOC(arena, OrenNayar)(r, sig));
```

9.2.2 PLASTICMATERIAL

Plastic can be modeled as a mixture of a diffuse and glossy scattering function with parameters controlling the particular colors and specular highlight size. The parameters to `PlasticMaterial` are two reflectivities, `Kd` and `Ks`, which respectively control the amounts of diffuse reflection and glossy specular reflection. Next is a roughness parameter (which ranges from zero to one) that determines the size of the specular highlight; the higher the roughness value, the larger the highlight. Figure 9.3 shows a plastic Killeroo. `PlasticMaterial` is defined in `materials/plastic.h` and `materials/plastic.cpp`.

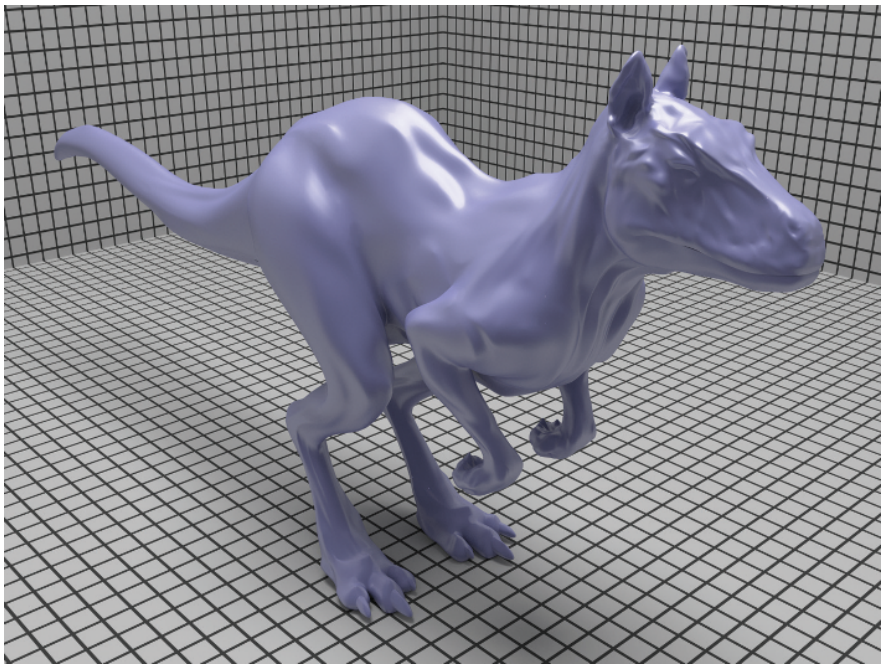


Figure 9.3: Killeroo Rendered with a Plastic Material. Note the combination of diffuse and glossy specular reflection. *(Killeroo courtesy of headus/Rezard.)*

BSDF::Add() 479
BSDF_ALLOC() 482
Lambertian 447
MatteMaterial::Kd 485
MatteMaterial::sigma 485
OrenNayar 449
PlasticMaterial 487
Spectrum 263
Spectrum::Clamp() 265
Texture::Evaluate() 520

⟨*PlasticMaterial Declarations*⟩ ≡
```
class PlasticMaterial : public Material {
public:
    ⟨PlasticMaterial Public Methods 487⟩
private:
    ⟨PlasticMaterial Private Data 487⟩
};
```

⟨*PlasticMaterial Public Methods*⟩ ≡ 487
```
PlasticMaterial(Reference<Texture<Spectrum> > kd,
                Reference<Texture<Spectrum> > ks,
                Reference<Texture<float> > rough,
                Reference<Texture<float> > bump)
    : Kd(kd), Ks(ks), roughness(rough), bumpMap(bump) {
}
```

⟨*PlasticMaterial Private Data*⟩ ≡ 487
```
Reference<Texture<Spectrum> > Kd, Ks;
Reference<Texture<float> > roughness, bumpMap;
```

The `PlasticMaterial::GetBSDF()` method follows the same basic structure as `MatteMaterial::GetBSDF()`: it evaluates textures, calls the bump-mapping function, allocates `BxDF`s, and initializes the `BSDF`.

⟨*PlasticMaterial Method Definitions*⟩ ≡
```
BSDF *PlasticMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                               const DifferentialGeometry &dgShading,
                               MemoryArena &arena) const {
    ⟨Allocate BSDF, possibly doing bump mapping with bumpMap 485⟩
    Spectrum kd = Kd->Evaluate(dgs).Clamp();
    BxDF *diff = BSDF_ALLOC(arena, Lambertian)(kd);
    Fresnel *fresnel = BSDF_ALLOC(arena, FresnelDielectric)(1.5f, 1.f);
    Spectrum ks = Ks->Evaluate(dgs).Clamp();
    float rough = roughness->Evaluate(dgs);
    BxDF *spec = BSDF_ALLOC(arena, Microfacet)
                 (ks, fresnel, BSDF_ALLOC(arena, Blinn)(1.f / rough));
    bsdf->Add(diff);
    bsdf->Add(spec);
    return bsdf;
}
```

Blinn 456
BSDF 478
BSDF::Add() 479
BxDF 428
DifferentialGeometry 102
Fresnel 436
FresnelDielectric 437
Lambertian 447
Material 483
MatteMaterial::GetBSDF() 485
MemoryArena 1015
Microfacet 454
MixMaterial 488
PlasticMaterial 487
PlasticMaterial::bumpMap 487
PlasticMaterial::GetBSDF() 487
PlasticMaterial::Kd 487
PlasticMaterial::Ks 487
PlasticMaterial::roughness 487
Reference 1011
Spectrum 263
Spectrum::Clamp() 265
Texture 519
Texture::Evaluate() 520

9.2.3 MIX MATERIAL

It's useful to be able to combine two `Materials` with a varying weight. The `MixMaterial` takes two other `Materials` and a `Spectrum`-valued texture; it uses the `Spectrum` returned by the texture to blend between the two materials at the point being shaded. It is defined in the files `materials/mixmat.h` and `materials/mixmat.cpp`.

⟨*MixMaterial Declarations*⟩ ≡
```
class MixMaterial : public Material {
public:
    ⟨MixMaterial Public Methods 488⟩
private:
    ⟨MixMaterial Private Data 488⟩
};
```

⟨*MixMaterial Public Methods*⟩ ≡ 488
```
MixMaterial(Reference<Material> mat1, Reference<Material> mat2,
            Reference<Texture<Spectrum> > sc)
    : m1(mat1), m2(mat2), scale(sc) {
}
```

⟨*MixMaterial Private Data*⟩ ≡ 488
```
Reference<Material> m1, m2;
Reference<Texture<Spectrum> > scale;
```

`MixMaterial::GetBSDF()` gets `BSDF *`s from the two materials. It scales `BxDF`s in the `BSDF` from the first material, `b1`, using the `ScaledBxDF` adapter class, and then scales the `BxDF`s from the second `BSDF`, adding them to `b1`. It may appear that there's a lurking memory leak in this code, in that the `BxDF *`s in `b1->bxdfs` are clobbered by newly allocated `ScaledBxDF`s. However, recall that those `BxDF`s, like the new ones here, were allocated through a `MemoryArena` and will thus be freed when the `MemoryArena` frees its entire block of memory.

⟨*MixMaterial Method Definitions*⟩ ≡
```
BSDF *MixMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                           const DifferentialGeometry &dgShading,
                           MemoryArena &arena) const {
    BSDF *b1 = m1->GetBSDF(dgGeom, dgShading, arena);
    BSDF *b2 = m2->GetBSDF(dgGeom, dgShading, arena);
    Spectrum s1 = scale->Evaluate(dgShading).Clamp();
    Spectrum s2 = (Spectrum(1.f) - s1).Clamp();
    int n1 = b1->NumComponents(), n2 = b2->NumComponents();
    for (int i = 0; i < n1; ++i)
        b1->bxdfs[i] = BSDF_ALLOC(arena, ScaledBxDF)(b1->bxdfs[i], s1);
    for (int i = 0; i < n2; ++i)
        b1->Add(BSDF_ALLOC(arena, ScaledBxDF)(b2->bxdfs[i], s2));
    return b1;
}
```

The implementation of `MixMaterial::GetBSDF()` needs direct access to the `bxdfs` member variables of the `BSDF` class. Because this is the only class that needs this access, we've just made `MixMaterial` a friend of `BSDF` rather than adding a number of accessor and setting methods.

⟨*BSDF Private Data*⟩ +≡ 478
```
friend class MixMaterial;
```

BSDF 478
BSDF::bxdfs 479
BSDF::NumComponents() 479
BxDF 428
DifferentialGeometry 102
Material 483
Material::GetBSDF() 483
MemoryArena 1015
MixMaterial 488
MixMaterial::m1 488
MixMaterial::m2 488
MixMaterial::scale 488
Reference 1011
ScaledBxDF 431
Spectrum 263
Spectrum::Clamp() 265
Texture 519
Texture::Evaluate() 520

9.2.4 MEASURED MATERIAL

The MeasuredMaterial class supports two forms of measured isotropic BRDF data: irregular and regular, corresponding to the BxDFs implemented in Sections 8.6.1 and 8.6.2. It is defined in the files materials/measured.h and materials/measured.cpp.

⟨*MeasuredMaterial Declarations*⟩ ≡
```
class MeasuredMaterial : public Material {
public:
    ⟨MeasuredMaterial Public Methods⟩
private:
    ⟨MeasuredMaterial Private Data 489⟩
};
```

For irregularly sampled BRDFs, the MeasuredMaterial constructor creates a kd-tree of the BRDF samples, using the BRDFRemap() function to map 4D pairs of directions to the 3D space that the IrregIsotropicBRDF uses for interpolation. For regularly sampled data, the table and its sampling resolution in the θ_h, θ_d, and ϕ_d dimensions are stored in corresponding member variables. (Only one of thetaPhiData or regularHalfangleData will be non-NULL.)

We won't include the code that reads this data from files here. The file formats for these two types of data are described in comments in the core/materials.cpp file.

⟨*MeasuredMaterial Private Data*⟩ ≡ 489
```
KdTree<IrregIsotropicBRDFSample> *thetaPhiData;
float *regularHalfangleData;
uint32_t nThetaH, nThetaD, nPhiD;
Reference<Texture<float> > bumpMap;
```

BRDFRemap() 465
BSDF 478
BSDF::Add() 479
BxDF 428
DifferentialGeometry 102
IrregIsotropicBRDF 464
IrregIsotropicBRDFSample 465
KdTree 1029
Material 483
MeasuredMaterial::nPhiD 489
MeasuredMaterial::nThetaD 489
MeasuredMaterial::nThetaH 489
MeasuredMaterial::regularHalfangleData 489
MeasuredMaterial::thetaPhiData 489
MemoryArena 1015
Reference 1011
RegularHalfangleBRDF 467
Texture 519

Once the data is in memory, the GetBSDF() method's task is straightforward. After the usual bump-mapping computation, it just has to allocate the appropriate BxDF implementation based on whether this instance of the MeasuredMaterial has regularly or irregularly sampled data.

⟨*MeasuredMaterial Method Definitions*⟩ ≡
```
BSDF *MeasuredMaterial::GetBSDF(const DifferentialGeometry &dgGeom,
                                const DifferentialGeometry &dgShading,
                                MemoryArena &arena) const {
    ⟨Allocate BSDF, possibly doing bump mapping with bumpMap 485⟩
    if (regularHalfangleData)
        bsdf->Add(BSDF_ALLOC(arena, RegularHalfangleBRDF)
            (regularHalfangleData, nThetaH, nThetaD, nPhiD));
    else if (thetaPhiData)
        bsdf->Add(BSDF_ALLOC(arena, IrregIsotropicBRDF)(thetaPhiData));
    return bsdf;
}
```

9.2.5 ADDITIONAL MATERIALS

Beyond these basic materials, there are eight more `Material` implementations available in `pbrt`, in the `materials/` directory. We will not show all of their implementations here, since they are all just variations on the basic themes introduced in the material implementations above. All take `Textures` that define scattering parameters, these textures are evaluated in their `GetBSDF()` methods, and appropriate `BxDFs` are created and returned in a `BSDF`. See the documentation on `pbrt`'s file format in the file `docs/fileformat.pdf` for a summary of the parameters that these materials take.

These materials include:

- `GlassMaterial`: Specular reflection and transmission, weighted by Fresnel terms for accurate angular-dependent variation.
- `MetalMaterial`: Metal, using the Fresnel equations for conductors. See the files in the directory `scenes/spds/metals/` for measured spectral data for the indices of refraction η and absorption coefficients k for a variety of metals.
- `MirrorMaterial`: A simple mirror, modeled with perfect specular reflection.
- `SubstrateMaterial`: A layered model that varies between glossy specular and diffuse reflection depending on the viewing angle (based on the `FresnelBlend` BRDF).
- `SubsurfaceMaterial` and `KdSubsurfaceMaterial`: Materials that return `BSSRDFs` that describe materials that exhibit subsurface scattering.
- `TranslucentMaterial`: A material that describes diffuse and glossy specular reflection and transmission through the surface.
- `UberMaterial`: A "kitchen sink" material representing the union of many of the preceding materials. This is a highly parameterized material that is particularly useful when converting scenes from other file formats into `pbrt`'s.

Figure 8.9 in the previous chapter shows the Killeroo model rendered with the glass material, and Figure 9.4 shows it with the `MetalMaterial`. Figure 9.5 demonstrates the `KdSubsurfaceMaterial`.

9.3 BUMP MAPPING

All of the `Materials` defined in the previous section take an optional `float` texture map that defines a displacement at each point on the surface: each point p has a displaced point p′ associated with it, defined by $\mathrm{p}' = \mathrm{p} + d(\mathrm{p})\mathbf{n}(\mathrm{p})$, where $d(\mathrm{p})$ is the offset returned by the displacement texture at p and $\mathbf{n}(\mathrm{p})$ is the surface normal at p (Figure 9.6). We would like to use this texture to compute shading normals so that the surface appears as if it actually had been offset by the displacement function, without modifying its geometry. This process is called *bump mapping*. For relatively small displacement functions, the visual effect of bump mapping can be quite convincing. This idea and the specific technique to compute these shading normals in a way that gives a plausible appearance of the actual displaced surface were developed by Blinn (1978).

Figure 9.7 shows the effect of applying bump mapping defined by an image map of a grid of lines to a sphere. A more complex example is shown in Figure 9.8, which shows a scene rendered with and without bump mapping. There, the bump map gives the appearance of a substantial amount of detail in the walls and floors that isn't actually present in

BSDF 478
BxDF 428
Material 483
Texture 519

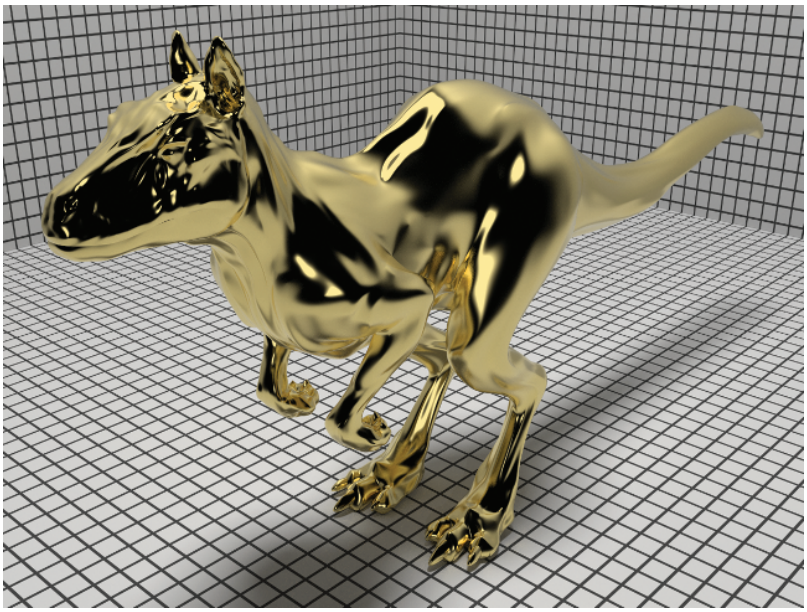


Figure 9.4: Killeroo rendered with the `MetalMaterial`, based on realistic measured gold scattering data. *(Killeroo courtesy of headus/Rezard.)*

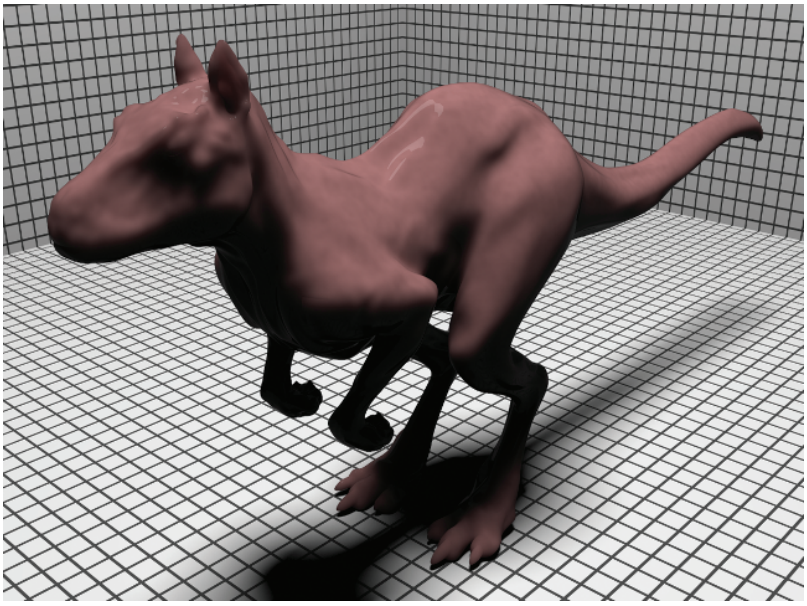


Figure 9.5: Killeroo rendered with the `KdSubsurfaceMaterial`, which models subsurface scattering (in conjunction with the subsurface scattering light transport techniques from Section 16.5). *(Killeroo courtesy of headus/Rezard.)*

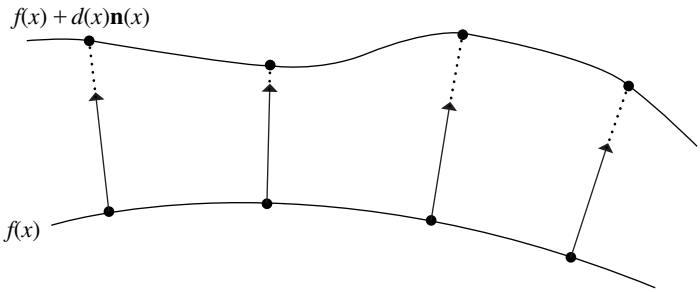


Figure 9.6: A displacement function associated with a material defines a new surface based on the old one, offset by the displacement amount along the normal at each point. `pbrt` doesn't compute a geometric representation of this displaced surface, but instead uses it to compute shading normals for bump mapping.

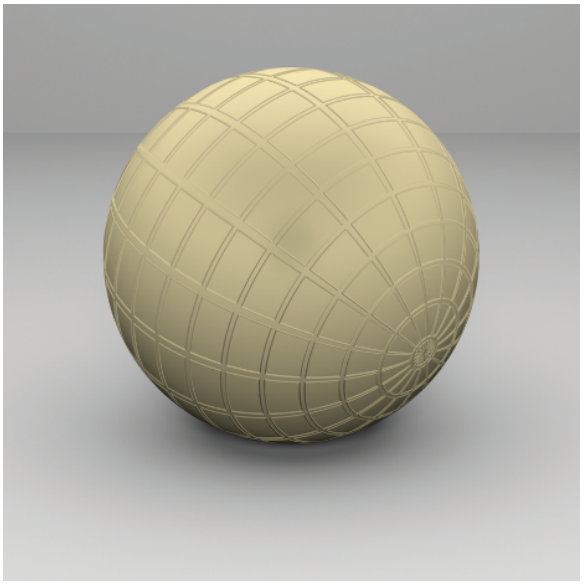


Figure 9.7: Using bump mapping to compute the shading normals for a sphere gives it the appearance of having much more geometric detail than is actually present.

the geometric model. Figure 9.9 shows one of the image maps used to define the bump function in Figure 9.8.

The `Material::Bump()` method is a utility routine for use by `Material` implementations. It is responsible for computing the effect of bump mapping at the point being shaded given a particular displacement `Texture`. So that future `Material` implementations aren't required to support bump mapping with this particular mechanism (or at all), we've

Material 483
Material::Bump() 495
Texture 519



(a)



(b)

Figure 9.8: The Sponza atrium model, rendered (a) without bump mapping and (b) with bump mapping. Bump mapping substantially increases the apparent geometric complexity of the model, without the increased rendering time and memory use that would result from a geometric representation with the equivalent amount of small-scale detail.



Figure 9.9: One of the image maps used as a bump map for the Sponza atrium rendering in Figure 9.8.

placed this method outside of the hard-coded material evaluation pipeline and left it as a function that particular material implementations can optionally call.

The implementation of `Material::Bump()` is based on finding an approximation to the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ of the displaced surface and using them in place of the surface's actual partial derivatives to compute the shading normal. (Recall that the surface normal is given by the cross product of these vectors, $\mathbf{n} = \partial \mathrm{p}/\partial u \times \partial \mathrm{p}/\partial v$.) Assume that the original surface is defined by a parametric function $p(u, v)$, and the bump offset function is a scalar function $d(u, v)$. Then the displaced surface is given by

$$p'(u, v) = p(u, v) + d(u, v)\mathbf{n}(u, v),$$

where $\mathbf{n}(u, v)$ is the surface normal at (u, v).

The partial derivatives of this function can be found using the chain rule. For example, the partial derivative in u is

$$\frac{\partial p'}{\partial u} = \frac{\partial p(u, v)}{\partial u} + \frac{\partial d(u, v)}{\partial u}\mathbf{n}(u, v) + d(u, v)\frac{\partial \mathbf{n}(u, v)}{\partial u}. \quad [9.1]$$

We already have computed the value of $\partial p(u, v)/\partial u$; it's $\partial \mathrm{p}/\partial u$ and is available in the `DifferentialGeometry` structure, which also stores the surface normal $\mathbf{n}(u, v)$ and the partial derivative $\partial \mathbf{n}(u, v)/\partial u = \partial \mathbf{n}/\partial u$. The displacement function $d(u, v)$ can be evaluated as needed, which leaves $\partial d(u, v)/\partial u$ as the only unknown term.

There are two possible approaches to finding the values of $\partial d(u, v)/\partial u$ and $\partial d(u, v)/\partial v$. One option would be to augment the `Texture` interface with a method to compute partial derivatives of the underlying texture function. For example, for image map textures mapped to the surface directly using its (u, v) parameterization, these partial derivatives can be computed by subtracting adjacent texels in the u and v directions. However, this approach is difficult to extend to complex procedural textures like some of the ones defined in Chapter 10. Therefore, `pbrt` directly computes these values with forward differencing in the `Material::Bump()` method, without modifying the `Texture` interface.

Recall the definition of the partial derivative:

$$\frac{\partial d(u, v)}{\partial u} = \lim_{\Delta_u \to 0} \frac{d(u + \Delta_u, v) - d(u, v)}{\Delta_u}.$$

Forward differencing approximates the value using a finite value of Δ_u and evaluating $d(u, v)$ at two positions. Thus, the final expression for $\partial p'/\partial u$ is the following (for simplicity, we have dropped the explicit dependence on (u, v) for some of the terms):

$$\frac{\partial p'}{\partial u} \approx \frac{\partial \mathrm{p}}{\partial u} + \frac{d(u + \Delta_u, v) - d(u, v)}{\Delta_u}\mathbf{n} + d(u, v)\frac{\partial \mathbf{n}}{\partial u}.$$

Interestingly enough, most bump-mapping implementations ignore the final term under the assumption that $d(u, v)$ is expected to be relatively small. (Since bump mapping is mostly useful for approximating small perturbations, this is a reasonable assumption.) The fact that many renderers do not compute the values $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ may also have something to do with this simplification. An implication of ignoring the last term is that the magnitude of the displacement function then does not affect the bump-mapped partial derivatives; adding a constant value to it globally doesn't affect the final result,

DifferentialGeometry 102
Material::Bump() 495
Texture 519

since only differences of the bump function affect it. pbrt computes all three terms since it has $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ readily available, although in practice this final term rarely makes a visually noticeable difference.

One important detail in the definition of this function is that the d parameter is declared to be of type const Reference<Texture<float> > &, rather than, for example, Reference<Texture<float> >. This is very important for performance, but for a subtle reason. If it was not a C++ reference type, then the Reference type would need to increment the Texture's reference count for the value passed on the stack, and the reference count would need to be decremented when the method returned. This isn't a problem for serial code, but with multiple threads of execution leads to a situation where multiple processing cores are modifying the same memory location whenever different rendering tasks have the same displacement texture passed to this method. This in turn leads to the expensive "read for ownership" operation described in Section A.9.1.

⟨*Material Method Definitions*⟩ ≡
```
void Material::Bump(const Reference<Texture<float> > &d,
                    const DifferentialGeometry &dgGeom,
                    const DifferentialGeometry &dgs,
                    DifferentialGeometry *dgBump) {
    ⟨Compute offset positions and evaluate displacement texture 495⟩
    ⟨Compute bump-mapped differential geometry 496⟩
    ⟨Orient shading normal to match geometric normal 496⟩
}
```

⟨*Compute offset positions and evaluate displacement texture*⟩ ≡ 495
```
DifferentialGeometry dgEval = dgs;
⟨Shift dgEval du in the u direction 495⟩
float uDisplace = d->Evaluate(dgEval);
⟨Shift dgEval dv in the v direction 496⟩
float vDisplace = d->Evaluate(dgEval);
float displace = d->Evaluate(dgs);
```

DifferentialGeometry 102
DifferentialGeometry::dndu 102
DifferentialGeometry::dpdu 102
DifferentialGeometry::dudx 505
DifferentialGeometry::dudy 505
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DifferentialGeometry::u 102
Material 483
Normal 65
Normal::Normalize() 65
Reference 1011
Texture 519
Texture::Evaluate() 520

One remaining issue is how to choose the offsets Δ_u and Δ_v for the finite differencing computations. They should be small enough that fine changes in $d(u, v)$ are captured but large enough so that available floating-point precision is sufficient to give a good result. Here, we will choose Δ_u and Δ_v values that lead to an offset that is about half the image space pixel sample spacing and use them to update the appropriate member variables in the DifferentialGeometry to reflect a shift to the offset position. (See Section 10.1.1 for an explanation of how the image space distances are computed.)

⟨*Shift* dgEval du *in the u direction*⟩ ≡ 495
```
float du = .5f * (fabsf(dgs.dudx) + fabsf(dgs.dudy));
if (du == 0.f) du = .01f;
dgEval.p = dgs.p + du * dgs.dpdu;
dgEval.u = dgs.u + du;
dgEval.nn = Normalize((Normal)Cross(dgs.dpdu, dgs.dpdv) +
                      du * dgs.dndu);
```

⟨*Shift* `dgEval dv` *in the v direction*⟩ ≡ 495
```
float dv = .5f * (fabsf(dgs.dvdx) + fabsf(dgs.dvdy));
if (dv == 0.f) dv = .01f;
dgEval.p = dgs.p + dv * dgs.dpdv;
dgEval.u = dgs.u;
dgEval.v = dgs.v + dv;
dgEval.nn = Normalize((Normal)Cross(dgs.dpdu, dgs.dpdv) +
                      dv * dgs.dndv);
```

Given the new positions and the displacement texture's values at them, the partial derivatives can be computed directly using Equation (9.1):

⟨*Compute bump-mapped differential geometry*⟩ ≡ 495
```
*dgBump = dgs;
dgBump->dpdu = dgs.dpdu + (uDisplace - displace) / du * Vector(dgs.nn) +
               displace * Vector(dgs.dndu);
dgBump->dpdv = dgs.dpdv + (vDisplace - displace) / dv * Vector(dgs.nn) +
               displace * Vector(dgs.dndv);
dgBump->nn = Normal(Normalize(Cross(dgBump->dpdu, dgBump->dpdv)));
if (dgs.shape->ReverseOrientation ^ dgs.shape->TransformSwapsHandedness)
    dgBump->nn *= -1.f;
```

Finally, this method flips the shading coordinate frame if needed, so that the shading normal lies in the hemisphere around the geometric normal. Since the shading normal represents a relatively small perturbation of the geometric normal, the two of them should always be in the same hemisphere.

⟨*Orient shading normal to match geometric normal*⟩ ≡ 495
```
dgBump->nn = Faceforward(dgBump->nn, dgGeom.nn);
```

Cross() 62
DifferentialGeometry::dndv 102
DifferentialGeometry::dpdu 102
DifferentialGeometry::dpdv 102
DifferentialGeometry::dvdx 505
DifferentialGeometry::dvdy 505
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DifferentialGeometry::u 102
DifferentialGeometry::v 102
Faceforward() 66
Normal 65
Normal::Normalize() 65
Shape::ReverseOrientation 108
Shape::TransformSwapsHandedness 108
Vector 57
Vector::Normalize() 63

FURTHER READING

Blinn (1978) invented the bump-mapping technique. Kajiya (1985) generalized the idea of bump mapping the normal to *frame mapping*, which also perturbs the surface's primary tangent vector and is useful for controlling the appearance of anisotropic reflection models. Mikkelsen's thesis (2008) carefully investigates a number of the assumptions underlying bump mapping, proposes generalizations, and also addresses a number of subtleties with respect to its application to real-time rendering.

Snyder and Barr (1987) noted the light leak problem from per-vertex shading normals and proposed a number of work-arounds. The method we have used in this chapter is from Section 5.3 of Veach's thesis (Veach 1997); it is a more robust solution than those of Snyder and Barr.

Shading normals introduce a number of subtle problems for physically based light transport algorithms that we have not addressed here. For example, they can easily lead to surfaces that reflect more energy than was incident upon them, which can wreak havoc with light transport algorithms that are designed under the assumption of energy conservation. Veach (1996) investigated this issue in depth and developed a number of solutions.

One visual shortcoming of bump mapping is that it doesn't naturally account for self-shadowing, where bumps cast shadows on the surface and prevent light from reaching nearby points. These shadows can have a significant impact on the appearance of rough surfaces. Max (1988) developed the *horizon mapping* technique, which performs a preprocess on bump maps stored in image maps to compute a term to account for this effect. This approach isn't directly applicable to procedural textures, however. Dana et al. (1999) have measured spatially varying reflection properties from real-world surfaces, including these self-shadowing effects; they convincingly demonstrate this effect's importance for accurate image synthesis.

Another difficult issue related to bump mapping is that antialiasing bump maps that have higher-frequency detail than can be represented in the image is quite difficult. In particular, it is not enough to remove high-frequency detail from the bump map function, but in general the BSDF needs to be modified to account for this detail. Fournier (1992) applied *normal distribution functions* to this problem, where the surface normal was generalized to represent a distribution of normal directions. Becker and Max (1993) developed algorithms for blending between bump maps and BRDFs that represented higher-frequency details. Schilling (1997, 2001) investigated this issue particularly for application to graphics hardware. More recently, effective approaches have been developed by Toksvig (2005) and Han et al. (2007).

An alternative to bump mapping is displacement mapping, where the bump function is used to actually modify the surface geometry, rather than just perturbing the normal (Cook 1984; Cook, Carpenter, and Catmull 1987). Advantages of displacement mapping include geometric detail on object silhouettes and the possibility of accounting for self-shadowing. Patterson and collaborators have described an innovative algorithm for displacement mapping with ray tracing where the geometry is unperturbed but the ray's direction is modified such that the intersections that are found are the same as would be found with the displaced geometry (Patterson, Hoggar, and Logie 1991; Logie and Patterson 1994). Heidrich and Seidel (1998) developed a technique for computing direct intersections with procedurally defined displacement functions.

With the advent of increased memory on computers and caching algorithms, the option of finely tessellating geometry and displacing its vertices for ray tracing has become feasible. Pharr and Hanrahan (1996) described an approach to this problem based on geometry caching, and Wang et al. (2000) described an adaptive tessellation algorithm that reduces memory requirements. Smits, Shirley, and Stark (2000) lazily tessellate individual triangles, saving a substantial amount of memory.

The book by Dorsey et al. (2008) provides comprehensive coverage of material and reflection modeling for computer graphics.

EXERCISES

`Material` 483
`PlasticMaterial::Kd` 487
`PlasticMaterial::Ks` 487
`Texture` 519

● 9.1 If the same `Texture` is bound to more than one component of a `Material` (for example, to both `PlasticMaterial::Kd` and `PlasticMaterial::Ks`), the texture will be evaluated twice. This unnecessarily duplicated work may lead to a noticeable increase in rendering time if the `Texture` is itself computationally expensive. Modify the materials in `pbrt` to eliminate this problem. Measure the

change in the system's performance, both for standard scenes as well as for contrived scenes that exhibit this redundancy.

● 9.2 One form of aliasing that pbrt doesn't try to eliminate is specular highlight aliasing. Glossy specular surfaces with high specular exponents, particularly if they have high curvature, are susceptible to aliasing as small changes in incident direction or surface position (and thus surface normal) may cause the highlight's contribution to change substantially. Read Amanatides's paper on this topic (Amanatides 1992) and extend pbrt to reduce specular aliasing, either using his technique or by developing your own. Most of the quantities needed to do the appropriate computations are already available—$\partial \mathbf{n}/\partial x$ and $\partial \mathbf{n}/\partial y$ in the DifferentialGeometry, and so on—although it will probably be necessary to extend the BxDF interface to provide more information about the values of specular exponents for glossy specular reflection components.

● 9.3 Another approach to specular highlight aliasing is to supersample the BSDF, evaluating it multiple times around the point being shaded. After reading the discussion of supersampling texture functions in Section 10.1, modify the BSDF::f() method to shift around to a set of positions around the intersection point but within the pixel sampling rate around the intersection point and evaluate the BSDF at each one of them when the BSDF evaluation routines are called. (Be sure to account for the change in normal using its partial derivatives.) How well does this approach combat specular highlight aliasing?

● 9.4 pbrt doesn't directly represent interfaces between objects that transmit light; instead, it assumes that for transmitted rays entering or leaving an object, the index of refraction outside the object is one. This assumption breaks down for many common cases—for example, a glass of water where rays pass directly from the glass into the water. Furthermore, in pbrt, if such a scene is modeled with the edge of the water coincident with the interior edge of the glass, the transmission from glass to water may be missed completely, since only one of the two ray intersections would be detected. Modify the system to eliminate this problem and render images showing that your approach renders correct images where pbrt currently computes an incorrect result. See, for example, Schmidt and Budge (2002) for one approach to this issue.

● 9.5 Read some of the papers on filtering bump maps referenced in the "Further Reading" section of this chapter, choose one of the techniques described there, and implement it in pbrt. Show both the visual artifacts from bump map aliasing without the technique you implement as well as examples of how well your implementation addresses them.

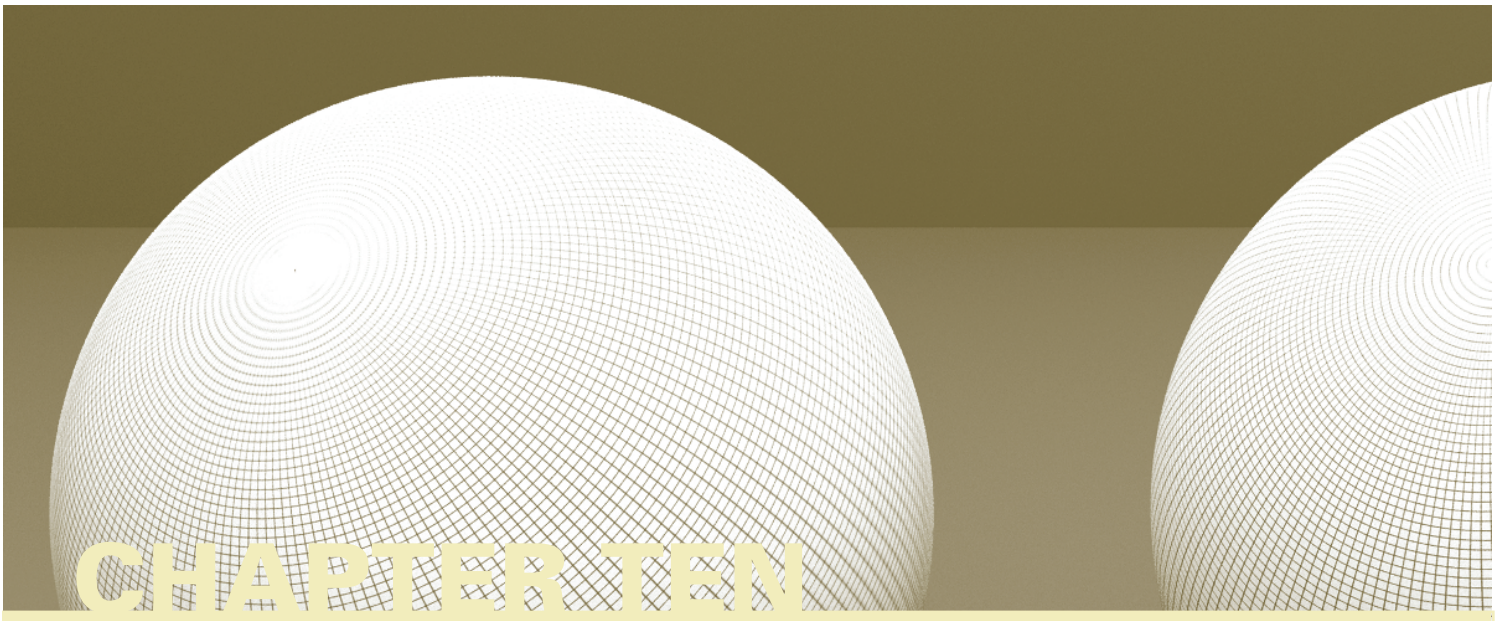
● 9.6 Neyret (1996, 1998) developed algorithms that automatically take descriptions of complex shapes and their reflective properties and turn them into generalized reflection models at different resolutions, each with limited frequency content. The advantage of this representation is that it makes it easy to select an appropriate level of detail for an object based on its size on the screen, thus reducing aliasing. Read these papers and implement the algorithms described

BSDF::f() 481
BxDF 428
DifferentialGeometry 102

in them in pbrt. Show how they can be used to reduce geometric aliasing from detailed geometry and extend them to address bump map aliasing.

● 9.7 Use the triangular face refinement infrastructure from the LoopSubdiv shape to implement displacement mapping in pbrt. The usual approach to displacement mapping is to finely tessellate the geometric shape and then to evaluate the displacement function at its vertices, moving each vertex the given distance along its normal. Because displacement mapping may make the extent of the shape larger, the bounding box of the undisplaced shape will need to be expanded by the maximum displacement distance that a particular displacement function will ever generate.

Refine each face of the mesh until, when projected onto the image, it is roughly the size of the separation between pixels. To do this, you will need to be able to estimate the image pixel-based length of an edge in the scene when it is projected onto the screen. Use the texturing infrastructure in Chapter 10 to evaluate displacement functions. See Patney et al. (2009) and Fisher et al. (2009) for discussion of issues related to avoiding cracks in the mesh due to adaptive tessellation.

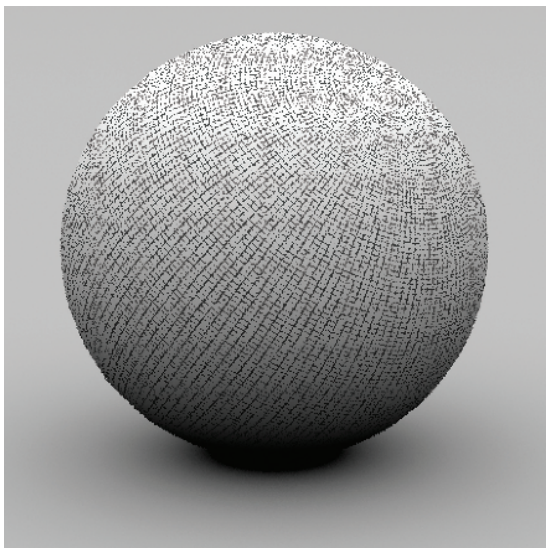

CHAPTER TEN

10 TEXTURE

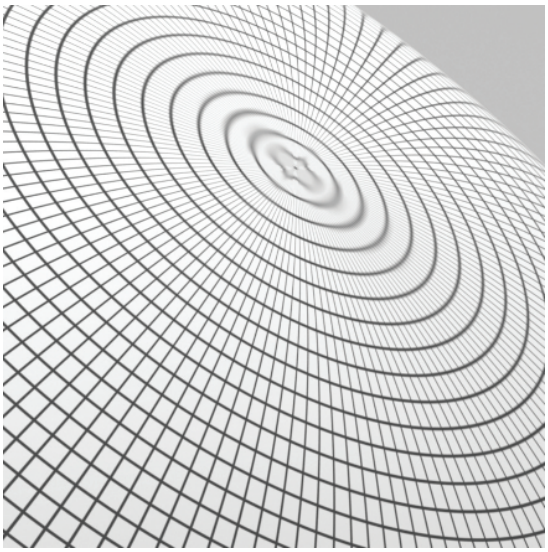
We will now describe a set of interfaces and classes that allow us to incorporate *texture* into our material models. Recall that the materials in Chapter 9 are all based on various parameters that describe their characteristics (diffuse reflectance, glossiness, etc.). Because real-world material properties typically vary over surfaces, it is necessary to be able to describe these patterns in some manner. In `pbrt`, because the texture abstractions are defined in a way that separates the pattern generation methods from the material implementations, it is easy to combine them in arbitrary ways, thereby making it easier to create a wide variety of appearances.

In `pbrt`, a texture is an extremely general concept: it is a function that maps points in some domain (e.g., a surface's (u, v) parametric space or (x, y, z) object space) to values in some other domain (e.g., spectra or the real numbers). A wide variety of implementations of texture classes are available in the system. For example, `pbrt` has textures that represent zero-dimensional functions that return a constant in order to accommodate surfaces that have the same parameter value everywhere. Image map textures are two-dimensional functions of (s, t) parameter values that use a 2D array of pixel values to compute values at a particular point (they are described in Section 10.4). There are even texture functions that compute values based on the values computed by other texture functions.

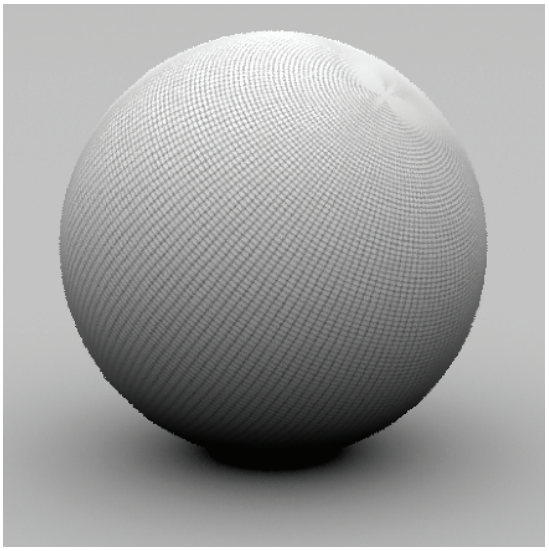
Textures may be a source of high-frequency variation in the final image. Figure 10.1 shows an image with severe aliasing due to a texture. Although the visual impact of this aliasing can be reduced with the nonuniform sampling techniques from Chapter 7, a better solution to this problem is to implement texture functions that adjust their frequency content based on the rate at which they are being sampled. For many texture functions, computing a reasonable approximation to the frequency content and antialiasing in this manner aren't too difficult and are substantially more efficient than reducing aliasing by increasing the image sampling rate.



(a)



(b)



(c)

Figure 10.1: Texture Aliasing. (a) An image of a grid texture on a sphere with one sample per pixel has severe aliasing artifacts. (b) A zoomed-in area from near the top of the sphere gives a sense of how much high-frequency detail is present between adjacent pixel sample positions. (c) The texture function has taken into account the image sampling rate to prefilter its function and remove high-frequency detail, resulting in an antialiased image, even with a single sample per pixel.

The first section of this chapter will discuss the problem of texture aliasing and general approaches to solving it. We will then describe the basic texture interface and illustrate its use with a few simple texture functions. Throughout the remainder of the chapter, we will present a variety of more complex texture implementations, demonstrating the use of a number of different texture antialiasing techniques along the way.

10.1 SAMPLING AND ANTIALIASING

The sampling task from Chapter 7 is a frustrating one since the aliasing problem was known to be unsolvable from the start. The infinite frequency content of geometric edges and hard shadows *guarantees* aliasing in the final images, no matter how high the image sampling rate. (Although the visual impact of this remaining aliasing can be reduced to be unobjectionable with a sufficient number of well-placed samples.) Fortunately, for textures things are not this difficult from the start: either there is often a convenient analytic form of the texture function available, which makes it possible to remove excessively high frequencies before sampling it, or it is possible to be careful when evaluating the function so as not to introduce high frequencies in the first place. When this problem is carefully addressed in texture implementations, as is done through the rest of this chapter, there is usually no need for more than one sample per pixel in order to render an image without texture aliasing.

Two problems must be addressed in order to remove aliasing from texture functions:

1. The sampling rate in texture space must be computed. The screen space sampling rate is known from the image resolution and pixel sampling rate, but here we need to determine the resulting sampling rate on a surface in the scene in order to find the rate at which the texture function is being sampled.
2. Given the texture sampling rate, sampling theory must be applied to guide the computation of a texture value that doesn't have higher-frequency variation than can be represented by the sampling rate (e.g., by removing excess frequencies beyond the Nyquist limit from the texture function).

These two issues will be addressed in turn throughout the rest of this section.

10.1.1 FINDING THE TEXTURE SAMPLING RATE

Consider an arbitrary texture function that is a function of position, $T(\mathrm{p})$, defined on a surface in the scene. If we ignore the complications introduced by visibility issues—the possibility that another object may occlude the surface at nearby image samples, or that the surface may have a limited extent on the image plane—this texture function can also be expressed as a function over points (x, y) on the image plane, $T(f(x, y))$, where $f(x, y)$ is the function that maps image points to points on the surface. Thus, $T(f(x, y))$ gives the value of the texture function as seen at image position (x, y).

As a simple example of this idea, consider a 2D texture function $T(s, t)$ applied to a quadrilateral that is perpendicular to the z axis and has corners at the world space points (0, 0, 0), (1, 0, 0), (1, 1, 0), and (0, 1, 0). If an orthographic camera is placed looking down the z axis such that the quadrilateral precisely fills the image plane and if points p

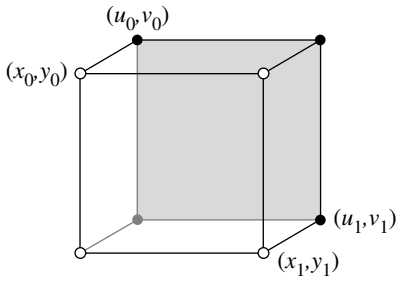


Figure 10.2: If a quadrilateral is viewed with an orthographic perspective such that the quadrilateral precisely fills the image plane, it's easy to compute the relationship between the sampling rate in (x, y) pixel coordinates and the texture sampling rate.

on the quadrilateral are mapped to 2D (s, t) texture coordinates by

$$s = \mathrm{p}_x \qquad t = \mathrm{p}_y,$$

then the relationship between (s, t) and screen (x, y) pixels is straightforward:

$$s = \frac{x}{x_r} \qquad t = \frac{y}{y_r},$$

where the overall image resolution is (x_r, y_r) (Figure 10.2). Thus, given a sample spacing of one pixel in the image plane, the sample spacing in (s, t) texture parameter space is $(1/x_r, 1/y_r)$, and the texture function must remove any detail at a higher frequency than can be represented at that sampling rate.

This relationship between pixel coordinates and texture coordinates, and thus the relationship between their sampling rates, is the key bit of information that determines the maximum frequency content allowable in the texture function. As a slightly more complex example, given a triangle with (s, t) texture coordinates at the vertices and viewed with a perspective projection, it's possible to analytically find the differences in s and t across the sample points on the image plane. This is the basis of basic texture map antialiasing in specialized graphics hardware.

For more complex scene geometry, camera projections, and mappings to texture coordinates, it is much more difficult to precisely determine the relationship between image positions and texture parameter values. Fortunately, for texture antialiasing, we don't need to be able to evaluate $f(x, y)$ for arbitrary (x, y), but just need to find the relationship between changes in pixel sample position and the resulting change in texture sample position at a particular point on the image. This relationship is given by the partial derivatives of this function, $\partial f/\partial x$ and $\partial f/\partial y$. For example, these can be used to find a first-order approximation to the value of f,

$$f(x', y') \approx f(x, y) + (x' - x)\frac{\partial f}{\partial x} + (y' - y)\frac{\partial f}{\partial y}.$$

If these partial derivatives are changing slowly with respect to the distances $x' - x$ and $y' - y$, this is a reasonable approximation. More importantly, the values of these partial derivatives give an approximation to the change in texture sample position for a shift of one pixel in the x and y directions, respectively, and thus directly yield the texture sampling rate. For example, in the previous quadrilateral example, $\partial s/\partial x = 1/x_r$, $\partial s/\partial y = 0$, $\partial t/\partial x = 0$, and $\partial t/\partial y = 1/y_r$.

The key to finding the values of these partial derivatives in the general case lies in the `RayDifferential` structure, which was defined in Section 2.5.1. This structure is initialized for each camera ray by the `Camera::GenerateRayDifferential()` method; it contains not only the ray actually being traced through the scene, but also two additional rays, one offset horizontally one pixel sample from the camera ray and the other offset vertically by one pixel sample. All of the geometric ray intersection routines use only the main camera ray for their computations; the auxiliary rays are ignored (this is easy to do because `RayDifferential` is a subclass of `Ray`).

Here we will use the offset rays to estimate the partial derivatives of the mapping $\mathrm{p}(x, y)$ from image position to world space position and the partial derivatives of the mappings $u(x, y)$ and $v(x, y)$ from (x, y) to (u, v) parametric coordinates, giving the partial derivatives of world space positions $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$ and the partial derivatives of (u, v) parametric coordinates $\partial u/\partial x$, $\partial v/\partial x$, $\partial u/\partial y$, and $\partial v/\partial y$. In Section 10.2, we will see how these can be used to compute the screen space derivatives of arbitrary quantities based on p or (u, v) and consequently the sampling rates of these quantities. The values of these partial derivatives at the intersection point are stored in the `DifferentialGeometry` structure. They are declared as `mutable`, since they are set in a method that takes a `const` differential geometry object.

Camera::
GenerateRayDifferential() 303
DifferentialGeometry 102
DifferentialGeometry::
ComputeDifferentials() 505
DifferentialGeometry::
dpdx 505
DifferentialGeometry::
dpdy 505
DifferentialGeometry::
dudx 505
DifferentialGeometry::
dudy 505
DifferentialGeometry::
dvdx 505
DifferentialGeometry::
dvdy 505
Intersection::GetBSDF() 484
Intersection::
GetBSSRDF() 484
Material 483
Ray 66
RayDifferential 69
RayDifferential::
hasDifferentials 69
Vector 57

⟨*DifferentialGeometry Public Data*⟩ +≡ 102
```
mutable Vector dpdx, dpdy;
mutable float dudx, dvdx, dudy, dvdy;
```

⟨*Initialize* `DifferentialGeometry` *from parameters*⟩ +≡ 102
```
dudx = dvdx = dudy = dvdy = 0;
```

The `DifferentialGeometry::ComputeDifferentials()` method computes these values. It is called by `Intersection::GetBSDF()` and `Intersection::GetBSSRDF()` before the `Material` `GetBSDF()` or `GetBSSRDF()` method is called so that these values will be available for any texture evaluation routines that are called by the material. Because ray differentials aren't available for all rays traced by the system (e.g., rays starting from light sources traced for photon mapping), the `hasDifferentials` field of the `RayDifferential` must be checked before these computations are done. If the differentials are not present, then the derivatives are all assumed to be zero (which will eventually lead to unfiltered point sampling of textures).

⟨*DifferentialGeometry Method Definitions*⟩ +≡
```
void DifferentialGeometry::ComputeDifferentials(
        const RayDifferential &ray) const {
    if (ray.hasDifferentials) {
        ⟨Estimate screen space change in p and (u, v) 507⟩
    }
```

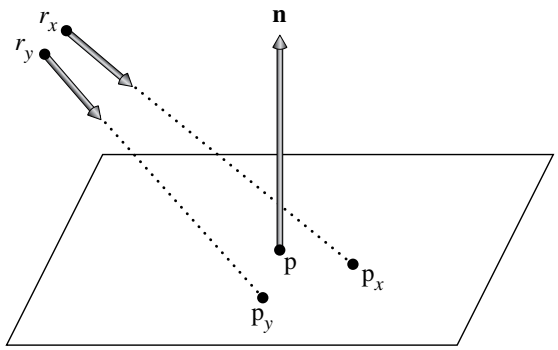


Figure 10.3: By approximating the local surface geometry at the intersection point with the tangent plane through p, approximations to the points at which the auxiliary rays r_x and r_y would intersect the surface can be found by finding their intersection points with the tangent plane p_x and p_y.

```
    else {
        dudx = dvdx = 0.;
        dudy = dvdy = 0.;
        dpdx = dpdy = Vector(0,0,0);
    }
}
```

The key to computing these estimates is the assumption that the surface is locally flat with respect to the sampling rate at the point being shaded. This is a reasonable approximation in practice, and it is hard to do much better. Because ray tracing is a point-sampling technique, we have no additional information about the scene in between the rays we traced. For highly curved surfaces or at silhouette edges, this approximation can break down, though it is rarely a source of noticeable error in practice. For this approximation, we need the plane through the point intersected by the main ray that is tangent to the surface. This plane is given by the implicit equation

$$ax + by + cz + d = 0,$$

where $a = \mathbf{n}_x$, $b = \mathbf{n}_y$, $c = \mathbf{n}_z$, and $d = -(\mathbf{n} \cdot \mathrm{p})$. We can then compute the intersection points p_x and p_y between the auxiliary rays r_x and r_y with this plane (Figure 10.3). These new points give an approximation to the partial derivatives of position on the surface $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$, based on forward differences:

$$\frac{\partial \mathrm{p}}{\partial x} \approx \mathrm{p}_x - \mathrm{p}, \quad \frac{\partial \mathrm{p}}{\partial y} \approx \mathrm{p}_y - \mathrm{p}.$$

Because the differential rays are offset one pixel sample in each direction, there's no need to divide these differences by a Δ value, since $\Delta = 1$.

⟨*Estimate screen space change in* p *and* (u, v)⟩ ≡ 505
```
⟨Compute auxiliary intersection points with plane 507⟩
dpdx = px - p;
dpdy = py - p;
⟨Compute (u, v) offsets at auxiliary points 508⟩
```

The ray–plane intersection algorithm gives the t value where a ray described by origin o and direction $\mathbf{d}$ intersects a plane described by $ax + by + cz + d = 0$:

$$t = \frac{-((a, b, c) \cdot \mathrm{o}) + d}{(a, b, c) \cdot \mathbf{d}}.$$

To compute this value for the two auxiliary rays, the plane's d coefficient is computed first. It isn't necessary to compute the a, b, and c coefficients, since they're available in `dg.nn`. We can then apply the formula directly.

⟨*Compute auxiliary intersection points with plane*⟩ ≡ 507
```
float d = -Dot(nn, Vector(p.x, p.y, p.z));
Vector rxv(ray.rxOrigin.x, ray.rxOrigin.y, ray.rxOrigin.z);
float tx = -(Dot(nn, rxv) + d) / Dot(nn, ray.rxDirection);
Point px = ray.rxOrigin + tx * ray.rxDirection;
Vector ryv(ray.ryOrigin.x, ray.ryOrigin.y, ray.ryOrigin.z);
float ty = -(Dot(nn, ryv) + d) / Dot(nn, ray.ryDirection);
Point py = ray.ryOrigin + ty * ray.ryDirection;
```

Using the positions p_x and p_y, an approximation to their respective (u, v) coordinates can be found by taking advantage of the fact that the surface's partial derivatives $\partial\mathrm{p}/\partial u$ and $\partial\mathrm{p}/\partial v$ form a (not necessarily orthogonal) coordinate system on the plane and that the coordinates of the auxiliary intersection points in terms of this coordinate system are their coordinates with respect to the (u, v) parameterization (Figure 10.4). Given

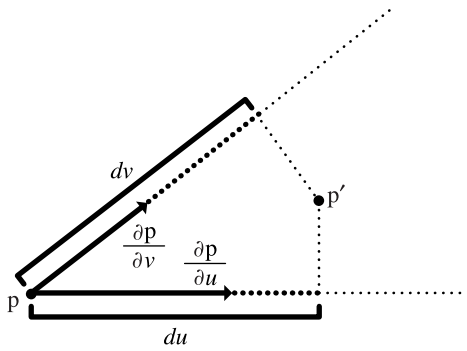


Figure 10.4: An estimate of the difference in (u, v) parametric coordinates from p to p′ can be found by finding the coordinates of p′ with respect to the coordinate system defined by p, $\partial\mathrm{p}/\partial u$, and $\partial\mathrm{p}/\partial v$.

DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Dot() 60
Point 63
RayDifferential::rxOrigin 69
Vector 57

a position p' on the plane, we can compute its position with respect to the coordinate system by

$$\mathrm{p}' = \mathrm{p} + \Delta_u \frac{\partial \mathrm{p}}{\partial u} + \Delta_v \frac{\partial \mathrm{p}}{\partial v},$$

or, equivalently,

$$\begin{pmatrix} \mathrm{p}'_x - \mathrm{p}_x \\ \mathrm{p}'_y - \mathrm{p}_y \\ \mathrm{p}'_z - \mathrm{p}_z \end{pmatrix} = \begin{pmatrix} \partial \mathrm{p}_x/\partial u & \partial \mathrm{p}_x/\partial v \\ \partial \mathrm{p}_y/\partial u & \partial \mathrm{p}_y/\partial v \\ \partial \mathrm{p}_z/\partial u & \partial \mathrm{p}_z/\partial v \end{pmatrix} \begin{pmatrix} \Delta_u \\ \Delta_v \end{pmatrix}.$$

A solution to this linear system of equations for one of the auxiliary points $\mathrm{p}' = \mathrm{p}_x$ or $\mathrm{p}' = \mathrm{p}_y$ gives the corresponding screen space partial derivatives $(\partial u/\partial x, \partial v/\partial x)$ or $(\partial u/\partial y, \partial v/\partial y)$, respectively.

This linear system has three equations with two unknowns—that is, it's overconstrained. We need to be careful since one of the equations may be degenerate—for example, if $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ are in the xy plane such that their z components are both zero, then the third equation will be degenerate. To deal with this case, because we only need two equations to solve the system, we'd like to choose the two that won't lead to a degenerate system. An easy way to do this is to take the cross product of $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$, see which coordinate of the result has the largest magnitude, and use the other two. Their cross product is already available in `nn`, so using this approach is straightforward. Even after all this, it may happen that the linear system has no solution (usually due to the partial derivatives not forming a coordinate system on the plane). In that case, all that can be done is to return arbitrary values.

⟨*Compute* (u, v) *offsets at auxiliary points*⟩ ≡ 507
```
⟨Initialize A, Bx, and By matrices for offset computation 508⟩
if (!SolveLinearSystem2x2(A, Bx, &dudx, &dvdx)) {
    dudx = 0.; dvdx = 0.;
}
if (!SolveLinearSystem2x2(A, By, &dudy, &dvdy)) {
    dudy = 0.; dvdy = 0.;
}
```

⟨*Initialize* `A`, `Bx`, *and* `By` *matrices for offset computation*⟩ ≡ 508
```
float A[2][2], Bx[2], By[2];
int axes[2];
if (fabsf(nn.x) > fabsf(nn.y) && fabsf(nn.x) > fabsf(nn.z)) {
    axes[0] = 1; axes[1] = 2;
}
else if (fabsf(nn.y) > fabsf(nn.z)) {
    axes[0] = 0; axes[1] = 2;
}
else {
    axes[0] = 0; axes[1] = 1;
}
⟨Initialize matrices for chosen projection plane 509⟩
```

DifferentialGeometry::dudx 505
DifferentialGeometry::dudy 505
DifferentialGeometry::dvdx 505
DifferentialGeometry::dvdy 505
DifferentialGeometry::nn 102
SolveLinearSystem2x2() 1020

⟨*Initialize matrices for chosen projection plane*⟩ ≡ 508

```
A[0][0] = dpdu[axes[0]];
A[0][1] = dpdv[axes[0]];
A[1][0] = dpdu[axes[1]];
A[1][1] = dpdv[axes[1]];
Bx[0] = px[axes[0]] - p[axes[0]];
Bx[1] = px[axes[1]] - p[axes[1]];
By[0] = py[axes[0]] - p[axes[0]];
By[1] = py[axes[1]] - p[axes[1]];
```

10.1.2 FILTERING TEXTURE FUNCTIONS

It is necessary to remove frequencies in texture functions that are past the Nyquist limit for the texture sampling rate. The goal is to compute, with as few approximations as possible, the result of the *ideal texture resampling* process, which says that in order to evaluate $T(f(x, y))$ without aliasing, we must first band-limit it, removing frequencies beyond the Nyquist limit by convolving it with the sinc filter:

$$T'_b(x, y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \operatorname{sinc}(x')\operatorname{sinc}(y')T'\left(f(x + x', y + y')\right) \mathrm{d}x'\mathrm{d}y'.$$

The band-limited function in turn should then be convolved with the pixel filter $g(x, y)$ centered at the (x, y) point on the screen at which we want to evaluate the texture function:

$$T'_f(x, y) = \int_{-\mathrm{xWidth}/2}^{\mathrm{xWidth}/2}\int_{-\mathrm{yWidth}/2}^{\mathrm{yWidth}/2} g(x', y')T'_b(x + x', y + y')\mathrm{d}x'\mathrm{d}y'.$$

This gives the theoretically perfect value for the texture as projected onto the screen.[1]

In practice, there are many simplifications that can be made to this process, with little reduction in visual quality. For example, a box filter may be used for the band-limiting step, and the second step is usually ignored completely, effectively acting as if the pixel filter were a box filter, which makes it possible to do the antialiasing work completely in texture space and simplifies the implementation significantly. The EWA filtering algorithm in Section 10.4.4 is a notable exception that doesn't make all of these simplifications.

Even the box filter, with all of its shortcomings, gives acceptable results for texture filtering in many cases. The box filter can be particularly easy to use, since it can be applied analytically by computing the average of the texture function over the appropriate region. Intuitively, this is a reasonable approach to the texture filtering problem, and it can be computed directly for many texture functions. Indeed, through the rest of this chapter, we will often use a box filter to average texture function values between samples and

DifferentialGeometry::
dpdu 102

DifferentialGeometry::
dpdv 102

DifferentialGeometry::p 102

MatteMaterial 484

1 One simplification that is present in this ideal filtering process is the implicit assumption that the texture function makes a linear contribution to frequency content in the image, so that filtering out its high frequencies removes high frequencies from the image. This is true for many uses of textures—for example, if an image map is used to modulate the diffuse term of a `MatteMaterial`. However, if a texture is used to determine the roughness of a glossy specular object, for example, this linearity assumption is incorrect, since the roughness value is both inverted and used as an exponent in a microfacet BRDF. We will ignore this issue here, since it isn't easily solved in general and usually doesn't cause substantial errors. The "Further Reading" section has more discussion of this issue.

informally use the term "filter region" to describe the area being averaged over. This is the most common approach when filtering texture functions.

An alternative to using the box filter to filter texture functions is to use the observation that the effect of the ideal sinc filter is to let frequency components below the Nyquist limit pass through unchanged but to remove frequencies past it. Therefore, if we know the frequency content of the texture function (e.g., if it is a sum of terms, each one with known frequency content), then if we replace the high-frequency terms with their average values, we are effectively doing the work of the sinc prefilter. This approach is known as *clamping* and is the basis for antialiasing in the textures based on the noise function in Section 10.6.

Finally, for texture functions where none of these techniques is easily applied, the approach of last resort is *supersampling*—the function is evaluated and filtered at multiple locations near the main evaluation point, thus increasing the sampling rate in texture space. If a box filter is used to filter these sample values, this is equivalent to averaging the value of the function. This approach can be expensive if the texture function is complex to evaluate, and as with image sampling a very large number of samples may be needed to remove aliasing. Although this is a brute-force solution, it is still more efficient than increasing the image sampling rate, since it doesn't incur the cost of tracing more rays through the scene.

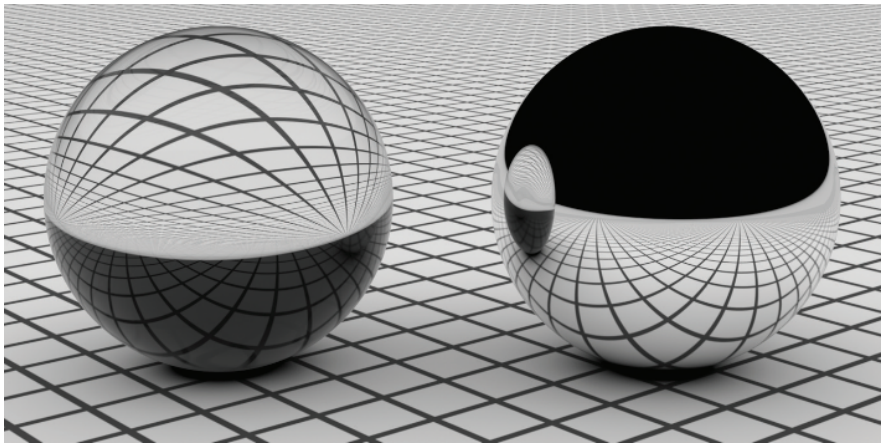
★ 10.1.3 RAY DIFFERENTIALS FOR SPECULAR REFLECTION AND TRANSMISSION

Given the effectiveness of ray differentials for finding filter regions for texture antialiasing for camera rays, it is useful to extend the method to make it possible to determine texture space sampling rates for objects that are seen indirectly via specular reflection or refraction; objects seen in mirrors, for example, should also no more have texture aliasing than directly visible objects. Igehy (1999) developed an elegant solution to the problem of how to find the appropriate differential rays for specular reflection and refraction, which is the approach used in `pbrt`.[2]

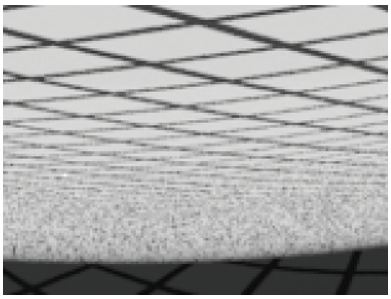
Figure 10.5 illustrates the difference that proper texture filtering for specular reflection and transmission can make. Figure 10.5(a) shows a glass ball and a mirrored ball on a plane with a texture map containing high-frequency components. Ray differentials ensure that the images of the texture seen via reflection and refraction from the balls are free of aliasing artifacts. A close-up view of the reflection in the glass ball is shown in Figure 10.5(b) and (c); Figure 10.5(b) was rendered without ray differentials for the reflected and transmitted rays, and Figure 10.5(c) was rendered with ray differentials. The aliasing errors in the left image are eliminated on the right without excessively blurring the texture.

In order to compute the reflected or transmitted ray differentials at a surface intersection point, we need an approximation to the rays that would have been traced at the intersec-

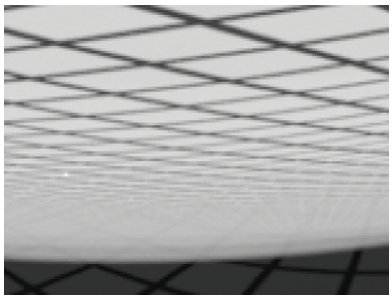
2 Igehy's formulation is slightly different than the one here—he effectively tracked the differences between the main ray and the offset rays, while we store the offset rays explicitly. The mathematics all work out to be the same in the end; we chose this alternative because we believe that it makes the algorithm's operation for camera rays easier to understand.



(a)



(b)



(c)

Figure 10.5: (a) Tracking ray differentials for reflected and refracted rays ensures that the image map texture seen in the balls is filtered to avoid aliasing. The left ball is glass, exhibiting reflection and refraction, and the right ball is a mirror, just showing reflection. Note that the texture is well filtered over both of the balls. (b) and (c) show a zoomed-in section of the glass ball; (b) shows the aliasing artifacts that are present if ray differentials aren't used, while (c) shows the result when they are.

tion points for the two offset rays in the ray differential that hit the surface (Figure 10.6). The new ray for the main ray is computed by the BSDF, so here we only need to compute the outgoing rays for the r_x and r_y differentials.

For both reflection and refraction, the origin of each differential ray is easily found. The `DifferentialGeometry::ComputeDifferentials()` method previously computed approximations for how much the surface position changes with respect to (x, y) position on the image plane $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$. Adding these offsets to the intersection point of the main ray gives approximate origins for the new rays. If the incident ray doesn't have differentials, then it's impossible to compute reflected ray differentials and this step is skipped.

DifferentialGeometry::
ComputeDifferentials() 505

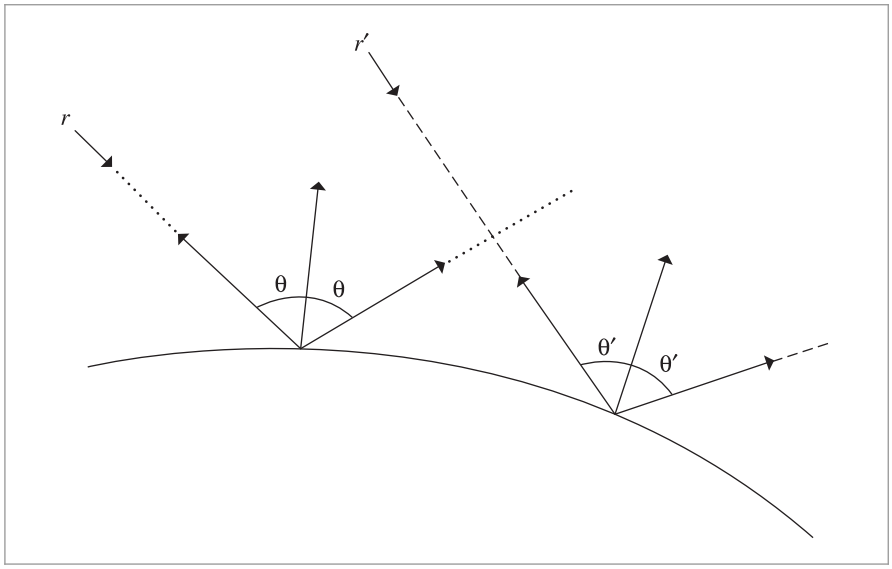


Figure 10.6: The specular reflection formula gives the direction of the reflected ray at a point on a surface. An offset ray for a ray differential (dashed line) will generally intersect the surface at a different point and be reflected in a different direction. The new direction is affected by both the different surface normal at the point as well as the offset ray's different incident direction. The computation to find the reflected direction for the offset ray in `pbrt` estimates the change in reflected direction as a function of image space position and approximates the ray differential's direction with the main ray's direction added to the estimated change in direction.

⟨*Compute ray differential* `rd` *for specular reflection*⟩ ≡ 46

```
RayDifferential rd(p, wi, ray, isect.rayEpsilon);
if (ray.hasDifferentials) {
    rd.hasDifferentials = true;
    rd.rxOrigin = p + isect.dg.dpdx;
    rd.ryOrigin = p + isect.dg.dpdy;
    ⟨Compute differential reflected directions 513⟩
}
```

Finding the directions of these rays is slightly more tricky. Igehy observed that if we know how much the reflected direction ω_i changes with respect to a shift of a pixel sample in the x and y directions on the image plane, we can use this information to approximate the direction of the offset rays:

$$\omega \approx \omega_i + \frac{\partial \omega_i}{\partial x}.$$

For a general world space surface normal and outgoing direction, the direction for perfect specular reflection is

$$\omega_i = -\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}.$$

Fortunately, the partial derivatives of this expression are easily computed:

DifferentialGeometry::
dpdx 505

DifferentialGeometry::
dpdy 505

Intersection::dg 186

Intersection::rayEpsilon 186

RayDifferential 69

RayDifferential::
hasDifferentials 69

$$\frac{\partial \omega_i}{\partial x} = \frac{\partial}{\partial x}\left(-\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}\right)$$
$$= -\frac{\partial \omega_o}{\partial x} + 2\left((\omega_o \cdot \mathbf{n})\frac{\partial \mathbf{n}}{\partial x} + \frac{\partial(\omega_o \cdot \mathbf{n})}{\partial x}\mathbf{n}\right).$$

Using the properties of the dot product, it can be shown that

$$\frac{\partial(\omega_o \cdot \mathbf{n})}{\partial x} = \frac{\partial \omega_o}{\partial x} \cdot \mathbf{n} + \omega_o \cdot \frac{\partial \mathbf{n}}{\partial x}.$$

The value of $\partial\omega_o/\partial x$ can be found from the difference between the direction of the ray differential's main ray and the direction of the r_x offset ray, and all of the other necessary quantities are readily available from the `DifferentialGeometry`, so the implementation of this computation for the partial derivatives in x and y is straightforward.

⟨*Compute differential reflected directions*⟩ ≡ 512

```
Normal dndx = bsdf->dgShading.dndu * bsdf->dgShading.dudx +
              bsdf->dgShading.dndv * bsdf->dgShading.dvdx;
Normal dndy = bsdf->dgShading.dndu * bsdf->dgShading.dudy +
              bsdf->dgShading.dndv * bsdf->dgShading.dvdy;
Vector dwodx = -ray.rxDirection - wo, dwody = -ray.ryDirection - wo;
float dDNdx = Dot(dwodx, n) + Dot(wo, dndx);
float dDNdy = Dot(dwody, n) + Dot(wo, dndy);
rd.rxDirection = wi - dwodx + 2 * Vector(Dot(wo, n) * dndx +
                                         dDNdx * n);
rd.ryDirection = wi - dwody + 2 * Vector(Dot(wo, n) * dndy +
                                         dDNdy * n);
```

A similar process of differentiating the equation for the direction of a specularly transmitted ray gives the equation to find the differential change in the transmitted direction. We won't include the derivation or our implementation here, but refer the interested reader to the original paper and to the source code, respectively.

10.2 TEXTURE COORDINATE GENERATION

DifferentialGeometry::
dndu 102

DifferentialGeometry::
dndv 102

DifferentialGeometry::
dudx 505

DifferentialGeometry::
dudy 505

DifferentialGeometry::
dvdx 505

DifferentialGeometry::
dvdy 505

Normal 65

TextureMapping2D 514

TextureMapping3D 519

Vector 57

Almost all of the textures in this chapter are functions that take a two-dimensional or three-dimensional coordinate and return a texture value. Sometimes there are obvious ways to choose these texture coordinates; for parametric surfaces, such as the quadrics in Chapter 3, there is a natural two-dimensional (u, v) parameterization of the surface, and for all surfaces the shading point p is a natural choice for a three-dimensional coordinate.

There is often no natural parameterization of complex surfaces, or the natural parameterization may be undesirable. For instance, the (u, v) values near the poles of spheres are severely distorted. Also, for an arbitrary subdivision surface, there is no simple, general-purpose way to assign texture values so that the entire $[0, 1]^2$ space is covered continuously and without distortion. In fact, creating smooth parameterizations of complex meshes with low distortion is an active area of research in computer graphics.

This section starts by introducing two abstract base classes—`TextureMapping2D` and `TextureMapping3D`—that provide an interface for computing these 2D and 3D texture

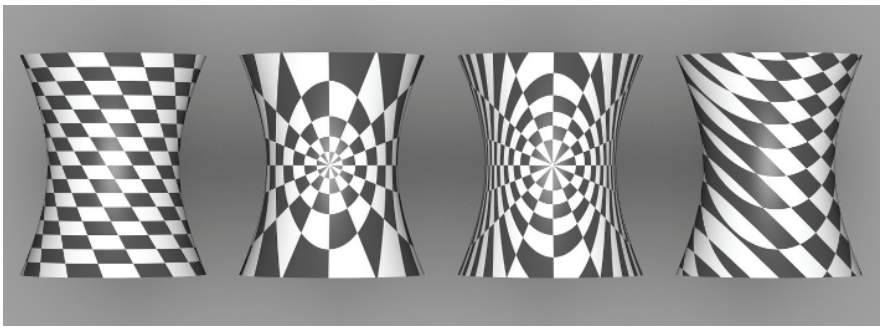


Figure 10.7: A checkerboard texture, applied to a hyperboloid with different texture coordinate generation techniques. From left to right, (u, v) mapping, spherical mapping, cylindrical mapping, and planar mapping.

coordinates. We will then implement a number of standard mappings using this interface (Figure 10.7 shows a number of them). `Texture` implementations store a pointer to a 2D or 3D mapping function as appropriate and use it to compute the texture coordinates at each point. Thus, it's easy to add new mappings to the system without having to modify all of the `Texture` implementations, and different mappings can be used for different textures associated with the same surface. In `pbrt`, we will use the convention that 2D texture coordinates are denoted by (s, t); this helps make clear the distinction between the intrinsic (u, v) parameterization of the underlying surface and the (possibly different) coordinate values used for texturing.

The `TextureMapping2D` base class has a single method, `TextureMapping2D::Map()`, which is given the `DifferentialGeometry` at the shading point and returns the (s, t) texture coordinates via `float` pointers. It also returns estimates for the change in s and t with respect to pixel x and y coordinates in the `dsdx`, `dtdx`, `dsdy`, and `dtdy` parameters so that textures that use the mapping can determine the (s, t) sampling rate and filter accordingly.

⟨*Texture Declarations*⟩ ≡
```
class TextureMapping2D {
public:
    ⟨TextureMapping2D Interface 514⟩
};
```

⟨*TextureMapping2D Interface*⟩ ≡ 514
```
virtual void Map(const DifferentialGeometry &dg,
                 float *s, float *t, float *dsdx, float *dtdx,
                 float *dtdy, float *dtdy) const = 0;
```

DifferentialGeometry 102
Texture 519
TextureMapping2D 514
TextureMapping2D::Map() 514

10.2.1 2D (u, v) MAPPING

The simplest mapping uses the 2D parametric (u, v) coordinates in the `DifferentialGeometry` to compute the texture coordinates. Their values can be offset and scaled with user-supplied values in each dimension.

⟨*Texture Declarations*⟩ +≡
```
class UVMapping2D : public TextureMapping2D {
public:
    ⟨UVMapping2D Public Methods⟩
private:
    float su, sv, du, dv;
};
```

⟨*Texture Method Definitions*⟩ ≡
```
UVMapping2D::UVMapping2D(float ssu, float ssv, float ddu, float ddv)
    : su(ssu), sv(ssv), du(ddu), dv(ddv) { }
```

The scale-and-shift computation to compute (s, t) coordinates is straightforward:

⟨*Texture Method Definitions*⟩ +≡
```
void UVMapping2D::Map(const DifferentialGeometry &dg,
                      float *s, float *t, float *dsdx, float *dtdx,
                      float *dsdy, float *dtdy) const {
    *s = su * dg.u + du;
    *t = sv * dg.v + dv;
    ⟨Compute texture differentials for 2D identity mapping 515⟩
}
```

Computing the differential change in s and t in terms of the original change in u and v and the scale amounts is also easy. Using the chain rule,

$$\frac{\partial s}{\partial x} = \frac{\partial u}{\partial x}\frac{\partial s}{\partial u} + \frac{\partial v}{\partial x}\frac{\partial s}{\partial v}$$

and similarly for the three other partial derivatives. From the mapping method,

$$s = s_u u + d_u,$$

so

$$\frac{\partial s}{\partial u} = s_u, \qquad \frac{\partial s}{\partial v} = 0,$$

and thus

$$\frac{\partial s}{\partial x} = s_u \frac{\partial u}{\partial x},$$

and so forth.

DifferentialGeometry 102
DifferentialGeometry::u 102
DifferentialGeometry::v 102
SphericalMapping2D 516
TextureMapping2D 514
UVMapping2D 515
UVMapping2D::du 515
UVMapping2D::dv 515
UVMapping2D::su 515
UVMapping2D::sv 515

⟨*Compute texture differentials for 2D identity mapping*⟩ ≡ 515
```
*dsdx = su * dg.dudx;
*dtdx = sv * dg.dvdx;
*dsdy = su * dg.dudy;
*dtdy = sv * dg.dvdy;
```

10.2.2 SPHERICAL MAPPING

Another useful mapping effectively wraps a sphere around the object. Each point is projected along the vector from the sphere's center through the point, up to the sphere's surface. There, the (u, v) mapping for the sphere shape is used. The SphericalMapping2D

stores a transformation that is applied to points before this mapping is performed; this effectively allows the mapping sphere to be arbitrarily positioned and oriented with respect to the object.

⟨*Texture Declarations*⟩ +≡
```
class SphericalMapping2D : public TextureMapping2D {
public:
    ⟨SphericalMapping2D Public Methods⟩
private:
    void sphere(const Point &P, float *s, float *t) const;
    Transform WorldToTexture;
};
```

⟨*Texture Method Definitions*⟩ +≡
```
void SphericalMapping2D::Map(const DifferentialGeometry &dg,
        float *s, float *t, float *dsdx, float *dtdx,
        float *dsdy, float *dtdy) const {
    sphere(dg.p, s, t);
    ⟨Compute texture coordinate differentials for sphere (u, v) mapping 517⟩
}
```

A short utility function computes the mapping for a single point. It will be useful to have this logic separated out for computing texture coordinate differentials.

⟨*Texture Method Definitions*⟩ +≡
```
void SphericalMapping2D::sphere(const Point &p, float *s, float *t) const {
    Vector vec = Normalize(WorldToTexture(p) - Point(0,0,0));
    float theta = SphericalTheta(vec);
    float phi = SphericalPhi(vec);
    *s = theta * INV_PI;
    *t = phi * INV_TWOPI;
}
```

We could use the chain rule again to compute the texture coordinate differentials, but will instead use a forward differencing approximation to demonstrate another way to compute these values that is useful for more complex mapping functions. Recall that the `DifferentialGeometry` stores the screen space partial derivatives $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$ that give the change in position as a function of change in image sample position. Therefore, if the s coordinate is computed by some function $f_s(\mathrm{p})$, it's easy to compute approximations like

$$\frac{\partial s}{\partial x} \approx \frac{f_s(\mathrm{p} + \Delta \partial \mathrm{p}/\partial x) - f_s(\mathrm{p})}{\Delta}.$$

As the distance Δ approaches 0, this gives the actual partial derivative at p.

One other detail is that the sphere mapping has a discontinuity in the mapping formula; there is a seam at $t = 1$, where the t texture coordinate discontinuously jumps back to zero. We can detect this case by checking to see if the value computed with forward differencing is greater than 0.5 and then adjusting it appropriately.

DifferentialGeometry 102
DifferentialGeometry::p 102
INV_PI 1002
INV_TWOPI 1002
Point 63
SphericalMapping2D 516
SphericalMapping2D::sphere() 516
SphericalMapping2D::WorldToTexture 516
SphericalPhi() 292
SphericalTheta() 292
TextureMapping2D 514
Transform 76
Vector 57
Vector::Normalize() 63

⟨*Compute texture coordinate differentials for sphere* (u, v) *mapping*⟩ ≡ 516
```
float sx, tx, sy, ty;
const float delta = .1f;
sphere(dg.p + delta * dg.dpdx, &sx, &tx);
*dsdx = (sx - *s) / delta;
*dtdx = (tx - *t) / delta;
if (*dtdx > .5) *dtdx = 1.f - *dtdx;
else if (*dtdx < -.5f) *dtdx = -(*dtdx + 1);
sphere(dg.p + delta * dg.dpdy, &sy, &ty);
*dsdy = (sy - *s) / delta;
*dtdy = (ty - *t) / delta;
if (*dtdy > .5) *dtdy = 1.f - *dtdy;
else if (*dtdy < -.5f) *dtdy = -(*dtdy + 1);
```

10.2.3 CYLINDRICAL MAPPING

The cylindrical mapping effectively wraps a cylinder around the object. It also supports a transformation to orient the mapping cylinder.

⟨*Texture Declarations*⟩ +≡
```
class CylindricalMapping2D : public TextureMapping2D {
public:
    ⟨CylindricalMapping2D Public Methods⟩
private:
    ⟨CylindricalMapping2D Private Methods 517⟩
    Transform WorldToTexture;
};
```

The cylindrical mapping has the same basic structure as the sphere mapping; just the mapping function is different. Therefore, we will omit the fragment that computes texture coordinate differentials, since it is essentially the same as the spherical version.

CylindricalMapping2D 517
CylindricalMapping2D::cylinder() 517
CylindricalMapping2D::WorldToTexture 517
DifferentialGeometry 102
DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
DifferentialGeometry::p 102
M_PI 1002
Point 63
TextureMapping2D 514
Transform 76
Vector 57
Vector::Normalize() 63

⟨*Texture Method Definitions*⟩ +≡
```
void CylindricalMapping2D::Map(const DifferentialGeometry &dg,
        float *s, float *t, float *dsdx, float *dtdx,
        float *dsdy, float *dtdy) const {
    cylinder(dg.p, s, t);
    ⟨Compute texture coordinate differentials for cylinder (u, v) mapping⟩
}
```

⟨*CylindricalMapping2D Private Methods*⟩ ≡ 517
```
void cylinder(const Point &p, float *s, float *t) const {
    Vector vec = Normalize(WorldToTexture(p) - Point(0,0,0));
    *s = (M_PI + atan2f(vec.y, vec.x)) / (2.f * M_PI);
    *t = vec.z;
}
```

10.2.4 PLANAR MAPPING

Another classic mapping method is planar mapping. The point is effectively projected onto a plane; a 2D parameterization of the plane then gives texture coordinates for the point. For example, a point p might be projected onto the $z = 0$ plane to yield texture coordinates given by $s = \mathrm{p}_x$ and $t = \mathrm{p}_y$.

In general, we can define such a parameterized plane with two nonparallel vectors $\mathbf{v}_s$ and $\mathbf{v}_t$ and offsets d_s and d_t. The texture coordinates are given by the coordinates of the point with respect to the plane's coordinate system, which are computed by taking the dot product of the vector from the point to the origin with each vector $\mathbf{v}_s$ and $\mathbf{v}_t$ and then adding the offset. For the example in the previous paragraph, we'd have $\mathbf{v}_s = (1, 0, 0)$, $\mathbf{v}_t = (0, 1, 0)$, and $d_s = d_t = 0$.

⟨*Texture Declarations*⟩ +≡
```
class PlanarMapping2D : public TextureMapping2D {
public:
    ⟨PlanarMapping2D Public Methods 518⟩
private:
    const Vector vs, vt;
    const float ds, dt;
};
```

⟨*PlanarMapping2D Public Methods*⟩ ≡ 518
```
PlanarMapping2D(const Vector &vv1, const Vector &vv2,
                float dds = 0, float ddt = 0)
    : vs(vv1), vt(vv2), ds(dds), dt(ddt) { }
```

The planar mapping differentials can be computed directly by finding the differentials of p in texture coordinate space.

⟨*Texture Method Definitions*⟩ +≡
```
void PlanarMapping2D::Map(const DifferentialGeometry &dg,
        float *s, float *t, float *dsdx, float *dtdx,
        float *dsdy, float *dtdy) const {
    Vector vec = dg.p - Point(0,0,0);
    *s = ds + Dot(vec, vs);
    *t = dt + Dot(vec, vt);
    *dsdx = Dot(dg.dpdx, vs);
    *dtdx = Dot(dg.dpdx, vt);
    *dsdy = Dot(dg.dpdy, vs);
    *dtdy = Dot(dg.dpdy, vt);
}
```

DifferentialGeometry 102
DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
DifferentialGeometry::p 102
Dot() 60
PlanarMapping2D 518
PlanarMapping2D::ds 518
PlanarMapping2D::dt 518
PlanarMapping2D::vs 518
PlanarMapping2D::vt 518
Point 63
TextureMapping2D 514
TextureMapping3D 519
Vector 57

10.2.5 3D MAPPING

We will also define a `TextureMapping3D` class that defines the interface for generating three-dimensional texture coordinates.

⟨*Texture Declarations*⟩ +≡
```
class TextureMapping3D {
public:
    ⟨TextureMapping3D Interface 519⟩
};
```

⟨*TextureMapping3D Interface*⟩ ≡ 519
```
virtual Point Map(const DifferentialGeometry &dg,
                  Vector *dpdx, Vector *dpdy) const = 0;
```

The natural three-dimensional mapping just takes the world space coordinate of the point and applies a linear transformation to it. This will often be a transformation that takes the point back to the primitive's object space.

⟨*Texture Declarations*⟩ +≡
```
class IdentityMapping3D : public TextureMapping3D {
public:
    IdentityMapping3D(const Transform &x)
        : WorldToTexture(x) { }
    Point Map(const DifferentialGeometry &dg, Vector *dpdx,
              Vector *dpdy) const;
private:
    Transform WorldToTexture;
};
```

Because a linear mapping is used, the differential change in texture coordinates can be found by applying the same mapping to the partial derivatives of position.

⟨*Texture Method Definitions*⟩ +≡
```
Point IdentityMapping3D::Map(const DifferentialGeometry &dg,
                             Vector *dpdx, Vector *dpdy) const {
    *dpdx = WorldToTexture(dg.dpdx);
    *dpdy = WorldToTexture(dg.dpdy);
    return WorldToTexture(dg.p);
}
```

DifferentialGeometry 102
DifferentialGeometry::dpdx 505
DifferentialGeometry::p 102
IdentityMapping3D 519
IdentityMapping3D::WorldToTexture 519
Point 63
ReferenceCounted 1010
Spectrum 263
Texture 519
TextureMapping3D 519
Transform 76
Vector 57

10.3 TEXTURE INTERFACE AND BASIC TEXTURES

`Texture` is a template class parameterized by the return type of its evaluation function. This design makes it possible to reuse almost all of the texturing code between textures that return different types. `pbrt` currently uses only `float` and `Spectrum` textures.

⟨*Texture Declarations*⟩ +≡
```
template <typename T> class Texture : public ReferenceCounted {
public:
    ⟨Texture Interface 520⟩
};
```

The key to Texture's interface is its evaluation function; it returns a value of the template type T. The only information it has access to in order to evaluate its value is the DifferentialGeometry at the point being shaded. Different textures in this chapter will use different parts of this structure to drive their evaluation.

⟨*Texture Interface*⟩ ≡ 519
```
virtual T Evaluate(const DifferentialGeometry &) const = 0;
```

10.3.1 CONSTANT TEXTURE

ConstantTexture returns the same value no matter where it is evaluated. Because it represents a constant function, it can be accurately reconstructed with any sampling rate and therefore needs no antialiasing. Although this texture is trivial, it is actually quite useful. By providing this class, all parameters to all Materials can be represented as Textures, whether they are spatially varying or not. For example, a red diffuse object will have a ConstantTexture that always returns red as the diffuse color of the material. This way, the shading system always evaluates a texture to get the surface properties at a point, avoiding the need for separate textured and nontextured versions of materials. This material's implementation is in textures/constant.h and textures/constant.cpp.

⟨*ConstantTexture Declarations*⟩ ≡
```
template <typename T> class ConstantTexture : public Texture<T> {
public:
    ⟨ConstantTexture Public Methods 520⟩
private:
    T value;
};
```

⟨*ConstantTexture Public Methods*⟩ ≡ 520
```
ConstantTexture(const T &v) { value = v; }
T Evaluate(const DifferentialGeometry &) const {
    return value;
}
```

10.3.2 SCALE TEXTURE

We have defined the texture interface in a way that makes it easy to use the output of one texture function when computing another. This is useful since it lets us define generic texture operations using any of the other texture types. The ScaleTexture takes two textures and returns the product of their values when evaluated. It is defined in textures/scale.h and textures/scale.cpp.

⟨*ScaleTexture Declarations*⟩ ≡
```
template <typename T1, typename T2>
class ScaleTexture : public Texture<T2> {
public:
    ⟨ScaleTexture Public Methods 521⟩
private:
    Reference<Texture<T1> > tex1;
    Reference<Texture<T2> > tex2;
};
```

ConstantTexture 520
DifferentialGeometry 102
Material 483
Reference 1011
ScaleTexture 520
Texture 519

⟨*ScaleTexture Public Methods*⟩ ≡ 520
```
ScaleTexture(Reference<Texture<T1> > t1, Reference<Texture<T2> > t2)
    : tex1(t1), tex2(t2) { }
```

`ScaleTexture` ignores antialiasing, leaving it to its two subtextures to antialias themselves but not making an effort to antialias their product. While it is easy to show that the product of two band-limited functions is also band limited, the maximum frequency present in the product may be greater than that of either of the two terms individually. Thus, even if the scale and value textures are perfectly antialiased, the result might not be. Fortunately, the most common kind of scale texture is a constant, in which case the other texture's antialiasing is sufficient.

⟨*ScaleTexture Public Methods*⟩ +≡ 520
```
T2 Evaluate(const DifferentialGeometry &dg) const {
    return tex1->Evaluate(dg) * tex2->Evaluate(dg);
}
```

10.3.3 MIX TEXTURES

The `MixTexture` class is a more general variation of `ScaleTexture`. It takes three textures as input: two may be of any type, and the third must return a floating-point value. The floating-point texture is then used to linearly interpolate between the two other textures. Note that a `ConstantTexture` could be used for the floating-point value to achieve a uniform blend, or a more complex `Texture` to blend in a spatially nonuniform way. This texture is defined in `textures/mix.h` and `textures/mix.cpp`.

⟨*MixTexture Declarations*⟩ ≡
```
template <typename T> class MixTexture : public Texture<T> {
public:
    ⟨MixTexture Public Methods 521⟩
private:
    Reference<Texture<T> > tex1, tex2;
    Reference<Texture<float> > amount;
};
```

⟨*MixTexture Public Methods*⟩ ≡ 521
```
MixTexture(Reference<Texture<T> > t1, Reference<Texture<T> > t2,
           Reference<Texture<float> > amt)
    : tex1(t1), tex2(t2), amount(amt) { }
```

To evaluate the mixture, the three textures are evaluated and the floating-point value is used to linearly interpolate between the two. When the blend amount (`amt`) is zero, the first texture's value is returned, and when it is one the second one's value is returned. We will generally assume that `amt` will be between zero and one, but this behavior is not enforced, so extrapolation is possible as well. As with the `ScaleTexture`, antialiasing is ignored, so the introduction of aliasing here is a possibility.

ConstantTexture 520
DifferentialGeometry 102
MixTexture 521
MixTexture::amount 521
MixTexture::tex1 521
MixTexture::tex2 521
Reference 1011
ScaleTexture 520
ScaleTexture::tex1 520
ScaleTexture::tex2 520
Texture 519
Texture::Evaluate() 520

⟨*MixTexture Public Methods*⟩ +≡ 521
```
T Evaluate(const DifferentialGeometry &dg) const {
    T t1 = tex1->Evaluate(dg), t2 = tex2->Evaluate(dg);
    float amt = amount->Evaluate(dg);
    return (1.f - amt) * t1 + amt * t2;
}
```

10.3.4 BILINEAR INTERPOLATION

⟨*BilerpTexture Declarations*⟩ ≡
```
template <typename T> class BilerpTexture : public Texture<T> {
public:
    ⟨BilerpTexture Public Methods 522⟩
private:
    ⟨BilerpTexture Private Data 522⟩
};
```

The `BilerpTexture` class provides bilinear interpolation between four constant values. Values are defined at (0, 0), (1, 0), (0, 1), and (1, 1) in (s, t) parameter space. The value at a particular (s, t) position is found by interpolating between them. It is defined in the files `textures/bilerp.h` and `textures/bilerp.cpp`.

⟨*BilerpTexture Public Methods*⟩ ≡ 522
```
BilerpTexture(TextureMapping2D *m, const T &t00, const T &t01,
              const T &t10, const T &t11)
    : mapping(m), v00(t00), v01(t01), v10(t10), v11(t11) {
}
```

⟨*BilerpTexture Private Data*⟩ ≡ 522
```
TextureMapping2D *mapping;
T v00, v01, v10, v11;
```

The interpolated value of the four values at an (s, t) position can be computed by three linear interpolations. For example, we can first use s to interpolate between the values at (0, 0) and (1, 0) and store that in a temporary `tmp1`. We can then do the same for the (0, 1) and (1, 1) values and store the result in `tmp2`. Finally, we use t to interpolate between `tmp1` and `tmp2` and obtain the final result. Mathematically, this is

$$\begin{aligned}\text{tmp}_1 &= (1-s)\text{v}_{00} + s\ \text{v}_{10}\\ \text{tmp}_2 &= (1-s)\text{v}_{01} + s\ \text{v}_{11}\\ \text{result} &= (1-t)\text{tmp}_1 + t\ \text{tmp}_2.\end{aligned}$$

Rather than storing the intermediate values explicitly, some algebraic rearrangement gives us the same result from an appropriately weighted average of the four corner values:

$$\text{result} = (1-s)(1-t)\text{v}_{00} + (1-s)t\ \text{v}_{01} + s(1-t)\text{v}_{10} + s\ t\ \text{v}_{11}.$$

BilerpTexture 522
BilerpTexture::mapping 522
BilerpTexture::v00 522
BilerpTexture::v01 522
BilerpTexture::v10 522
BilerpTexture::v11 522
DifferentialGeometry 102
Texture 519
Texture::Evaluate() 520
TextureMapping2D 514

⟨*BilerpTexture Public Methods*⟩ +≡ 522

```
T Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    return (1-s)*(1-t) * v00 + (1-s)*(  t) * v01 +
           (  s)*(1-t) * v10 + (  s)*(  t) * v11;
}
```

10.4 IMAGE TEXTURE

The `ImageTexture` class stores a 2D array of point-sampled values of a texture function. It uses these samples to reconstruct a continuous image function that can be evaluated at an arbitrary (s, t) position. These sample values are often called *texels*, since they are similar to pixels in an image but are used in the context of a texture. Image textures are the most widely used type of texture in computer graphics; digital photographs, scanned artwork, images created with image-editing programs, and images generated by renderers are all extremely useful sources of data for this particular texture representation (Figure 10.8). The term *texture map* is often used to refer to this type of texture, although this usage blurs the distinction between the mapping that computes texture coordinates and the texture function itself. The implementation of this texture is in the files `textures/imagemap.h` and `textures/imagemap.cpp`.



Figure 10.8: An Example of Image Textures. Here, a rendered image is used to modulate the diffuse color of the sphere, making the sphere look painted.

BilerpTexture::mapping 522
BilerpTexture::v00 522
BilerpTexture::v01 522
BilerpTexture::v10 522
BilerpTexture::v11 522
DifferentialGeometry 102
TextureMapping2D::Map() 514

The ImageTexture class is different from other textures in the system in that it is parameterized on both the data type of the texels it stores in memory as well as the data type of the value that it returns. Making this distinction allows us to create, for example, ImageTextures that store RGBSpectrum values in memory, but always return Spectrum values. In this way, when the system is compiled with full-spectral rendering enabled, the memory cost to store full SampledSpectrum texels doesn't necessarily need to be paid.

⟨*ImageTexture Declarations*⟩ ≡
```
template <typename Tmemory, typename Treturn>
    class ImageTexture : public Texture<Treturn> {
public:
    ⟨ImageTexture Public Methods 527⟩
private:
    ⟨ImageTexture Private Methods 526⟩
    ⟨ImageTexture Private Data 524⟩
};
```

The caller provides the ImageTexture with the filename of an image map, parameters that control the filtering of the map for antialiasing, and parameters that make it possible to scale and gamma-correct the texture values. The scale and gamma parameters will be explained later in this section, and the texture filtering parameters will be explained in Section 10.4.2. The contents of the file are used to create an instance of the MIPMap class that stores the texels in memory and handles the details of reconstruction and filtering to reduce aliasing.

For an ImageTexture that stores RGBSpectrum values in memory, its MIPMap stores the image data using three floating-point values for each sample. This can be a somewhat wasteful representation, since a single image map may have millions of texels and may not need the full 32 bits of accuracy from the floats used to store RGB values for each of them. Exercise 10.1 at the end of this chapter discusses this issue further.

⟨*ImageTexture Method Definitions*⟩ ≡
```
template <typename Tmemory, typename Treturn>
ImageTexture<Tmemory, Treturn>::ImageTexture(TextureMapping2D *m,
        const string &filename, bool doTrilinear, float maxAniso,
        ImageWrap wrapMode, float scale, float gamma) {
    mapping = m;
    mipmap = GetTexture(filename, doTrilinear, maxAniso,
                        wrapMode, scale, gamma);
}
```

⟨*ImageTexture Private Data*⟩ ≡ 524
```
MIPMap<Tmemory> *mipmap;
TextureMapping2D *mapping;
```

ImageTexture 524
ImageTexture::
 GetTexture() 525
ImageTexture::mapping 524
ImageTexture::mipmap 524
ImageWrap 530
MIPMap 530
RGBSpectrum 279
SampledSpectrum 266
Spectrum 263
Texture 519
TextureMapping2D 514

10.4.1 TEXTURE CACHING

An image map may require a significant amount of memory; because the user may reuse a texture many times within a scene, pbrt maintains a table of image maps that have been loaded so far, so that they are only loaded into memory once even if they

are used in more than one ImageTexture. The ImageTexture constructor calls the static ImageTexture::GetTexture() method to get a MIPMap representation of the desired texture. If the image map does need to be loaded from disk, ReadImage() handles the low-level details of this process and returns an array of texel values.

⟨*ImageTexture Method Definitions*⟩ +≡
```
template <typename Tmemory, typename Treturn> MIPMap<Tmemory> *
ImageTexture<Tmemory, Treturn>::GetTexture(const string &filename,
        bool doTrilinear, float maxAniso, ImageWrap wrap,
        float scale, float gamma) {
    ⟨Look for texture in texture cache 525⟩
    int width, height;
    RGBSpectrum *texels = ReadImage(filename, &width, &height);
    MIPMap<Tmemory> *ret = NULL;
    if (texels) {
        ⟨Convert texels to type Tmemory and create MIPMap 526⟩
    }
    else {
        ⟨Create one-valued MIPMap 527⟩
    }
    textures[texInfo] = ret;
    return ret;
}
```

TexInfo is a simple structure that holds the image map's filename and filtering parameters; all of these must match for a MIPMap to be reused in another ImageTexture. Its definition is straightforward and won't be included here.

⟨*ImageTexture Private Data*⟩ +≡ 524
```
static std::map<TexInfo, MIPMap<Tmemory> *> textures;
```

⟨*ImageTexture Method Definitions*⟩ +≡
```
template <typename Tmemory, typename Treturn>
    std::map<TexInfo,
             MIPMap<Tmemory> *> ImageTexture<Tmemory, Treturn>::textures;
```

⟨*Look for texture in texture cache*⟩ ≡ 525
```
TexInfo texInfo(filename, doTrilinear, maxAniso, wrap, scale, gamma);
if (textures.find(texInfo) != textures.end())
    return textures[texInfo];
```

Because the image-loading routine returns an array of RGBSpectrum values for the texels, it is necessary to convert these values to the particular type Tmemory of texel that this MIPMap is storing (e.g., float) if the type of Tmemory isn't RGBSpectrum. The per-texel conversion is handled by the utility routine ImageTexture::convertIn(). This conversion

ImageTexture 524
ImageTexture::convertIn() 526
ImageTexture::GetTexture() 525
ImageTexture::textures 525
ImageWrap 530
MIPMap 530
ReadImage() 1004
RGBSpectrum 279
TexInfo 525

is wasted work in the common case where the MIPMap is storing RGBSpectrum values, but the flexibility it gives is worth this relatively small cost in efficiency.[3]

⟨*Convert texels to type* Tmemory *and create* MIPMap⟩ ≡ 525
```
Tmemory *convertedTexels = new Tmemory[width*height];
for (int i = 0; i < width*height; ++i)
    convertIn(texels[i], &convertedTexels[i], scale, gamma);
ret = new MIPMap<Tmemory>(width, height, convertedTexels, doTrilinear,
                          maxAniso, wrap);
delete[] texels;
delete[] convertedTexels;
```

Per-texel conversion is done using C++ function overloading. For every type to which we would like to be able to convert these values, a separate ImageTexture::convertIn() function must be provided. In the loop over texels earlier, C++'s function overloading mechanism will select the appropriate instance of ImageTexture::convertIn() based on the destination type. Unfortunately, it is not possible to return the converted value from the function, since C++ doesn't support overloading by return type.

In addition to converting types, these functions optionally scale and gamma correct the texel values to map them to a desired range. Gamma correction is particularly important to handle carefully: computer displays have the property that most of them don't exhibit a linear relationship between the pixel values to be displayed and the radiance that they emit. Thus, an artist may create a texture map where, as seen on an LCD display, one part of the image appears twice as bright as another. However, the corresponding pixel values won't in fact have a 2:1 relationship. (Conversely, pixels that do have a 2:1 relationship don't lead to pixels with a 2:1 brightness ratio.)

This discrepancy is a problem for a renderer using such an image as a texture map, since the renderer usually expects a linear relationship between texel values and the quantity that they represent. In practice, the relationship between pixel values and display brightness is well modeled with a power curve with the exponent of a value gamma. (Gamma values between 1.5 and 2.2 are typical.) *Gamma correction*, as performed in convertIn(), can be used to restore an approximately linear relationship between texel values and the values they are intended to represent. See the "Further Reading" section for more discussion of gamma correction.

⟨*ImageTexture Private Methods*⟩ ≡ 524
```
static void convertIn(const RGBSpectrum &from, RGBSpectrum *to,
                      float scale, float gamma) {
    *to = Pow(scale * from, gamma);
}
static void convertIn(const RGBSpectrum &from, float *to,
                      float scale, float gamma) {
    *to = powf(scale * from.y(), gamma);
}
```

ImageTexture::convertIn() 526
MIPMap 530
RGBSpectrum 279
Spectrum 263
Spectrum::Pow() 265
Spectrum::y() 273

3 Additional C++ template trickery could ensure that this step was skipped when T is type Spectrum if the cost of this unnecessary work was unacceptable.

If the texture file wasn't found or was unreadable, an image map with a single sample with a value of one is created so that the renderer can continue to generate an image of the scene without needing to abort execution. The `ReadImage()` function will issue a warning message in this case.

⟨*Create one-valued* `MIPMap`⟩ ≡ 525
```
Tmemory *oneVal = new Tmemory[1];
oneVal[0] = powf(scale, gamma);
ret = new MIPMap<Tmemory>(1, 1, oneVal);
delete[] oneVal;
```

After the image is rendered and the system is cleaning up, the `ClearCache()` method is called to free the memory for the entries in the texture cache.

⟨*ImageTexture Public Methods*⟩ ≡ 524
```
static void ClearCache() {
    typename std::map<TexInfo, MIPMap<Tmemory> *>::iterator iter;
    iter = textures.begin();
    while (iter != textures.end()) {
        delete iter->second;
        ++iter;
    }
    textures.erase(textures.begin(), textures.end());
}
```

The `ImageTexture::Evaluate()` routine does the usual texture coordinate computation and then hands the image map lookup to the `MIPMap`, which does the image filtering work for antialiasing. The returned value is still of type `Tmemory`; another conversion step similar to `ImageTexture::convertIn()` above converts to the returned type `Treturn`.

⟨*ImageTexture Method Definitions*⟩ +≡
```
template <typename Tmemory, typename Treturn> Treturn
ImageTexture<Tmemory,
             Treturn>::Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    Tmemory mem = mipmap->Lookup(s, t, dsdx, dtdx, dsdy, dtdy);
    Treturn ret;
    convertOut(mem, &ret);
    return ret;
}
```

⟨*ImageTexture Private Methods*⟩ +≡ 524
```
static void convertOut(const RGBSpectrum &from, Spectrum *to) {
    float rgb[3];
    from.ToRGB(rgb);
    *to = Spectrum::FromRGB(rgb);
}
static void convertOut(float from, float *to) {
    *to = from;
}
```

DifferentialGeometry 102
ImageTexture 524
ImageTexture::convertIn() 526
ImageTexture::Evaluate() 527
ImageTexture::mapping 524
ImageTexture::mipmap 524
ImageTexture::textures 525
MIPMap 530
MIPMap::Lookup() 540
ReadImage() 1004
RGBSpectrum 279
Spectrum 263
Spectrum::FromRGB() 277
Spectrum::ToRGB() 275
TexInfo 525
TextureMapping2D::Map() 514

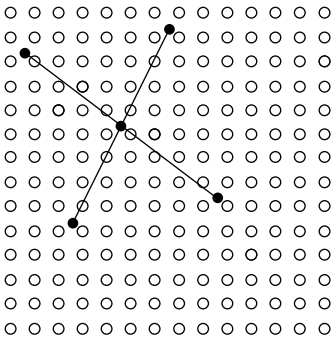


Figure 10.9: Given a point at which to perform an image map (denoted by the solid point in the center) and estimates of the texture space sampling rate (denoted by adjacent solid points), it may be necessary to filter the contributions of a large number of texels in the image map (denoted by open points).

10.4.2 MIP MAPS

As always, if the image function has higher-frequency detail than can be represented by the texture sampling rate, aliasing will be present in the final image. Any frequencies higher than the Nyquist limit must be removed by prefiltering before the function is evaluated. Figure 10.9 shows the basic problem we face: an image texture has texels that are samples of some image function at a fixed frequency. The filter region for the lookup is given by its (s, t) center point and offsets to the estimated texture coordinate locations for the adjacent image samples. Because these offsets are estimates of the texture sampling rate, we must remove any frequencies higher than twice the distance to the adjacent samples in order to satisfy the Nyquist criterion.

The texture sampling and reconstruction process has a few key differences from the image sampling process discussed in Chapter 7. These differences make it possible to address the antialiasing problem with more effective and less computationally expensive techniques. For example, here it is inexpensive to get the value of a sample—only an array lookup is necessary (as opposed to having to trace a ray). Further, because the texture image function is fully defined by the set of samples and there is no mystery about what its highest frequency could be, there is no uncertainty related to the function's behavior between samples. These differences make it possible to remove detail from the texture before sampling, thus eliminating aliasing.

However, the texture sampling rate will typically change from pixel to pixel. The sampling rate is determined by scene geometry and its orientation, the texture coordinate mapping function, and the camera projection and image sampling rate. Because the sampling rate is not fixed, texture filtering algorithms need to be able to filter over arbitrary regions of texture samples efficiently.

MIPMap 530

The `MIPMap` class implements two methods for efficient texture filtering with spatially varying filter widths. The first, trilinear interpolation, is fast and easy to implement and

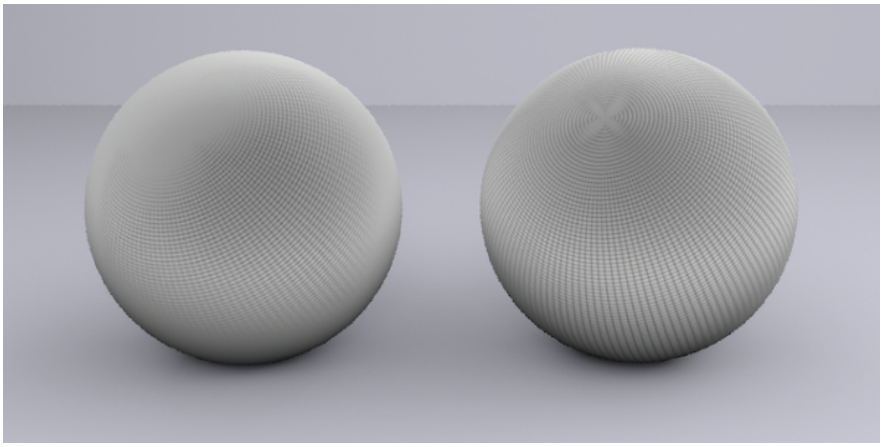


Figure 10.10: Filtering the image map properly substantially improves the image. On the left, trilinear interpolation was used; on the right, the EWA algorithm. Both of these approaches give a much better image than the unfiltered image map in Figure 11.1. Trilinear interpolation is less able to handle strongly anisotropic filter footprints than EWA, which is why the edges of the sphere on the left are a uniform gray color (the overall average value of the texture), while the edges of the sphere on the right are much better able to continue to resolve detail from the image map before fading to gray.

has been widely used for texture filtering in graphics hardware. The second, elliptically weighted averaging, is slower and more complex, but returns extremely high-quality results. Figure 10.1 shows the aliasing errors that result from ignoring texture filtering and just bilinearly interpolating texels from the most detailed level of the image map. Figure 10.10 shows the improvement from using the triangle filter and the EWA algorithm instead.

To limit the potential number of texels that need to be accessed, both of these filtering methods use an *image pyramid* of increasingly lower-resolution prefiltered versions of the original image to accelerate their operation.[4] The original image texels are at the bottom level of the pyramid, and the image at each level is half the resolution of the previous level, up to the top level, which has a single texel representing the average of all of the texels in the original image. This collection of images needs at most 1/3 more memory than storing the most detailed level alone and can be used to quickly find filtered values over large regions of the original image. The basic idea behind the pyramid is that if a large area of texels needs to be filtered a reasonable approximation is to use a higher level of the pyramid and do the filtering over the same area there, accessing many fewer texels.

`MIPMap` is a template class and is parameterized by the data type of the image texels. `pbrt` creates `MIPMaps` of both `RGBSpectrum` and `float` images; `float` MIP maps are used for representing directional distributions of intensity from goniometric light sources, for example. The `MIPMap` implementation requires that the type `T` support just a few basic operations, including addition and multiplication by a scalar.

`MIPMap` 530
`RGBSpectrum` 279

4 The name "MIP map" comes from the Latin *multum in parvo*, which means "much in little," a nod to the image pyramid.

The ImageWrap enumerant, passed to the MIPMap constructor, specifies the desired behavior when the supplied texture coordinates are not in the legal [0, 1] range.

⟨*MIPMap Declarations*⟩ ≡
```
typedef enum {
    TEXTURE_REPEAT,
    TEXTURE_BLACK,
    TEXTURE_CLAMP
} ImageWrap;
```

⟨*MIPMap Declarations*⟩ +≡
```
template <typename T> class MIPMap {
public:
    ⟨MIPMap Public Methods 535⟩
private:
    ⟨MIPMap Private Methods 532⟩
    ⟨MIPMap Private Data 530⟩
};
```

In the constructor, the MIPMap copies the image data provided by the caller, resizes the image if necessary to ensure that its resolution is a power of two in each direction, and initializes a lookup table used by the elliptically weighted average filtering method in Section 10.4.4. It also records the desired behavior for texture coordinates that fall outside of the legal range in the wrapmode argument.

⟨*MIPMap Method Definitions*⟩ ≡
```
template <typename T>
MIPMap<T>::MIPMap(uint32_t sres, uint32_t tres, const T *img, bool doTri,
                  float maxAniso, ImageWrap wm) {
    doTrilinear = doTri;
    maxAnisotropy = maxAniso;
    wrapMode = wm;
    T *resampledImage = NULL;
    if (!IsPowerOf2(sres) || !IsPowerOf2(tres)) {
        ⟨Resample image to power-of-two resolution 531⟩
    }
    width = sres;
    height = tres;
    ⟨Initialize levels of MIPMap from image 534⟩
    if (resampledImage) delete[] resampledImage;
    ⟨Initialize EWA filter weights if needed 544⟩
}
```

⟨*MIPMap Private Data*⟩ ≡ 530
```
bool doTrilinear;
float maxAnisotropy;
ImageWrap wrapMode;
```

ImageWrap 530
IsPowerOf2() 1001
MIPMap 530

Implementation of an image pyramid is somewhat easier if the resolution of the original image is an exact power of two in each direction; this ensures that there is a straightfor-

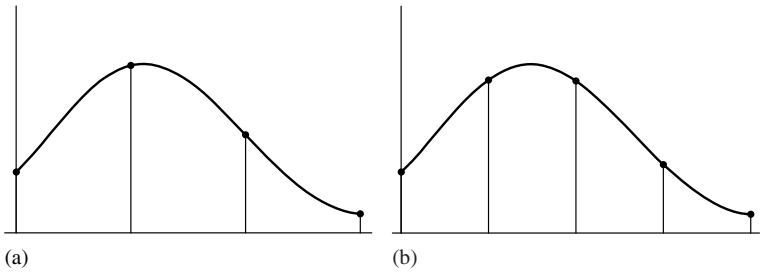


Figure 10.11: To increase an image's resolution to be a power of two, the `MIPMap` performs two 1D resampling steps with a separable reconstruction filter. (a) A 1D function reconstructed from four samples, denoted by dots. (b) To represent the same image function with more samples, we just need to reconstruct the continuous function and evaluate it at the new positions.

ward relationship between the level of the pyramid and the number of texels at that level. If the user has provided an image where the resolution in one or both of the dimensions is not a power of two, then the `MIPMap` constructor starts by resizing the image up to the next power-of-two resolution greater than the original resolution before constructing the pyramid. Exercise 10.5 at the end of the chapter describes an approach to building image pyramids with non-power-of-two resolutions.

Image resizing here involves more application of the sampling and reconstruction theory from Chapter 7: we have an image function that has been sampled at one sampling rate, and we'd like to reconstruct a continuous image function from the original samples to resample at a new set of sample positions. Because this represents an increase in the sampling rate from the original rate, we don't have to worry about introducing aliasing due to undersampling high-frequency components in this step; we only need to reconstruct and directly resample the new function. Figure 10.11 illustrates this task in 1D.

The `MIPMap` uses a separable reconstruction filter for this task; recall from Section 7.7 that separable filters can be written as the product of one-dimensional filters: $f(x, y) = f(x)f(y)$. One advantage of using a separable filter is that if we are using one to resample an image from one resolution (s, t) to another (s', t'), then we can implement the resampling as two one-dimensional resampling steps, first resampling in s to create an image of resolution (s', t) and then resampling that image to create the final image of resolution (s', t'). Resampling the image via two 1D steps in this manner simplifies implementation and makes the number of texels accessed for each texel in the final image a linear function of the filter width, rather than a quadratic one.

⟨*Resample image to power-of-two resolution*⟩ ≡ 530

```
uint32_t sPow2 = RoundUpPow2(sres), tPow2 = RoundUpPow2(tres);
⟨Resample image in s direction 532⟩
⟨Resample image in t direction⟩
sres = sPow2;
tres = tPow2;
```

MIPMap 530
RoundUpPow2() 1002

Reconstructing the original image function and sampling it at a new texel's position is mathematically equivalent to centering the reconstruction filter kernel at the new texel's position and weighting the nearby texels in the original image appropriately. Thus, each new texel is a weighted average of a small number of texels in the original image.

The `MIPMap::resampleWeights()` method determines which original texels contribute to each new texel and what the values are of the contribution weights for each new texel. It returns the values in an array of `ResampleWeight` structures for all of the texels in a 1D row or column of the image. Because this information is the same for all rows of the image when resampling in s and all columns when resampling in t, it's more efficient to compute it once for each of the two passes and then reuse it many times for each one. Given these weights, the image is first magnified in the s direction, turning the original image with resolution `(sres,tres)` into an image with resolution `(sPow2,tres)`, which is stored in `resampledImage`. The implementation here allocates enough space in `resampledImage` to hold the final zoomed image with resolution `(sPow2,tPow2)`, so two large allocations can be avoided.

⟨*Resample image in s direction*⟩ ≡ 531
```
ResampleWeight *sWeights = resampleWeights(sres, sPow2);
resampledImage = new T[sPow2 * tPow2];
⟨Apply sWeights to zoom in s direction 534⟩
delete[] sWeights;
```

For the reconstruction filter used here, no more than four of the original texels will contribute to each new texel after zooming, so `ResampleWeight` only needs to hold four weights. Because the four texels are contiguous, we only store the offset to the first one.

⟨*MIPMap Private Data*⟩ +≡ 530
```
struct ResampleWeight {
    int firstTexel;
    float weight[4];
};
```

⟨*MIPMap Private Methods*⟩ ≡ 530
```
ResampleWeight *resampleWeights(uint32_t oldres, uint32_t newres) {
    Assert(newres >= oldres);
    ResampleWeight *wt = new ResampleWeight[newres];
    float filterwidth = 2.f;
    for (uint32_t i = 0; i < newres; ++i) {
        ⟨Compute image resampling weights for ith texel 533⟩
    }
    return wt;
}
```

Just as it was important to distinguish between discrete and continuous pixel coordinates in Chapter 7, the same issues need to be addressed with texel coordinates here. We will use the same conventions as described in Section 7.1.7. For each new texel, this function starts by computing its continuous coordinates in terms of the old texel coordinates. This value is stored in `center`, because it is the center of the reconstruction filter for the new texel. Next, it is necessary to find the offset to the first texel that contributes to the new texel.

Assert() 1005
MIPMap::
resampleWeights() 532
ResampleWeight 532

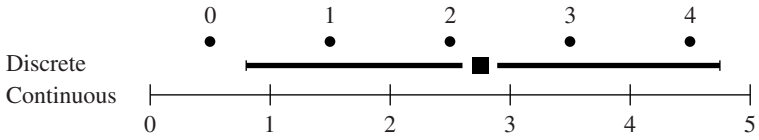


Figure 10.12: The computation to find the first texel inside a reconstruction filter's support is slightly tricky. Consider a filter centered around continuous coordinate 2.75 with width 2, as shown here. The filter's support covers the range [0.75, 4.75], although texel zero is outside the filter's support: adding 0.5 to the lower end before taking the floor to find the discrete texel gives the correct starting texel, number one.

This is a slightly tricky calculation—after subtracting the filter width to find the start of the filter's nonzero range, it is necessary to add an extra 0.5 offset to the continuous coordinate before taking the floor to find the discrete coordinate. Figure 10.12 illustrates why this offset is needed.

Starting from this first contributing texel, this function loops over four texels, computing each one's offset to the center of the filter kernel and the corresponding filter weight. The reconstruction filter function used to compute the weights, `Lanczos()`, is the same as the one in `LanczosSincFilter::Sinc1D()`.

⟨*Compute image resampling weights for* `i`*th texel*⟩ ≡ 532
```
float center = (i + .5f) * oldres / newres;
wt[i].firstTexel = Floor2Int((center - filterwidth) + 0.5f);
for (int j = 0; j < 4; ++j) {
    float pos = wt[i].firstTexel + j + .5f;
    wt[i].weight[j] = Lanczos((pos - center) / filterwidth);
}
⟨Normalize filter weights for texel resampling 533⟩
```

Depending on the filter function used, the four filter weights may not sum to one. Therefore, to ensure that the resampled image won't be any brighter or darker than the original image, the weights are normalized here.

⟨*Normalize filter weights for texel resampling*⟩ ≡ 533
```
float invSumWts = 1.f / (wt[i].weight[0] + wt[i].weight[1] +
                         wt[i].weight[2] + wt[i].weight[3]);
for (uint32_t j = 0; j < 4; ++j)
    wt[i].weight[j] *= invSumWts;
```

Floor2Int() 1002
Lanczos() 533
LanczosSincFilter::
 Sinc1D() 402
ResampleWeight::
 firstTexel 532
ResampleWeight::weight 532

⟨*Texture Declarations*⟩ +≡
```
float Lanczos(float, float tau=2);
```

Once the weights have been computed, it's easy to apply them to compute the zoomed texels. For each of the `tres` horizontal scan lines in the original image, a pass is made across the `sPow2` texels in the s-zoomed image using the precomputed weights to compute their values.

⟨*Apply* sWeights *to zoom in s direction*⟩ ≡ 532
```
for (uint32_t t = 0; t < tres; ++t) {
    for (uint32_t s = 0; s < sPow2; ++s) {
        ⟨Compute texel (s, t) in s-zoomed image 534⟩
    }
}
```

The ImageWrap parameter to the MIPMap constructor determines the convention to be used for out-of-bounds texel coordinates. It either remaps them to valid values with a modulus or clamp calculation or uses a black texel value.

⟨*Compute texel* (*s*, *t*) *in s-zoomed image*⟩ ≡ 534
```
resampledImage[t*sPow2+s] = 0.;
for (int j = 0; j < 4; ++j) {
    int origS = sWeights[s].firstTexel + j;
    if (wrapMode == TEXTURE_REPEAT)
        origS = Mod(origS, sres);
    else if (wrapMode == TEXTURE_CLAMP)
        origS = Clamp(origS, 0, sres-1);
    if (origS >= 0 && origS < (int)sres)
        resampledImage[t*sPow2+s] += sWeights[s].weight[j] *
                                     img[t*sres + origS];
}
```

The process for resampling in the t direction is almost the same as for s, so we won't include the implementation here. Once we have an image with resolutions that are powers of two, the levels of the MIP map can be initialized, starting from the bottom (finest) level. Each higher level is found by filtering the texels from the previous level.

Because image maps use a fair amount of memory, and because 8 to 20 texels are typically used per image texture lookup to compute a filtered value, it's worth carefully considering how the texels are laid out in memory, since reducing cache misses while accessing the texture map can noticeably improve the renderer's performance. Because both of the texture filtering methods implemented in this section access a set of texels in a rectangular region of the image map each time a lookup is performed, the MIPMap uses the BlockedArray template class to store the 2D arrays of texel values, rather than using a standard C++ array. The BlockedArray reorders the array values in memory in a way that improves cache coherence when the values are accessed with these kinds of rectangular patterns; it is described in Section A.5.5 in Appendix A.

⟨*Initialize levels of MIPMap from image*⟩ ≡ 530
```
nLevels = 1 + Log2Int(float(max(sres, tres)));
pyramid = new BlockedArray<T> *[nLevels];
⟨Initialize most detailed level of MIPMap 535⟩
for (uint32_t i = 1; i < nLevels; ++i) {
    ⟨Initialize ith MIPMap level from i − 1st level 536⟩
}
```

BlockedArray 1017
Clamp() 1000
ImageWrap 530
Log2Int() 1001
MIPMap 530
Mod() 1001
ResampleWeight::firstTexel 532
ResampleWeight::weight 532
TEXTURE_CLAMP 530
TEXTURE_REPEAT 530

⟨*MIPMap Private Data*⟩ +≡ 530
```
BlockedArray<T> **pyramid;
uint32_t width, height, nLevels;
```

⟨*MIPMap Public Methods*⟩ ≡ 530
```
uint32_t Width() const { return width; }
uint32_t Height() const { return height; }
uint32_t Levels() const { return nLevels; }
```

The base level of the MIP map, which holds the original data (or the resampled data, if it didn't originally have power-of-two resolutions), is initialized by the default `BlockedArray` constructor.

⟨*Initialize most detailed level of MIPMap*⟩ ≡ 534
```
pyramid[0] = new BlockedArray<T>(sres, tres, img);
```

Before showing how the rest of the levels are initialized, we will first define a texel access function that will be used during that process. `MIPMap::Texel()` returns a reference to the texel value for the given discrete integer-valued texel position. As described earlier, if an out-of-range texel coordinate is passed in, this method effectively repeats the texture over the entire 2D texture coordinate domain by taking the modulus of the coordinate with respect to the texture size, clamps the coordinates to the valid range so that the border pixels are used, or returns a black texel for out-of-bounds coordinates.

⟨*MIPMap Method Definitions*⟩ +≡
```
template <typename T>
const T &MIPMap<T>::Texel(uint32_t level, int s, int t) const {
    const BlockedArray<T> &l = *pyramid[level];
    ⟨Compute texel (s, t) accounting for boundary conditions 535⟩
    return l(s, t);
}
```

⟨*Compute texel* (*s*, *t*) *accounting for boundary conditions*⟩ ≡ 535
```
switch (wrapMode) {
    case TEXTURE_REPEAT:
        s = Mod(s, l.uSize());
        t = Mod(t, l.vSize());
        break;
    case TEXTURE_CLAMP:
        s = Clamp(s, 0, l.uSize() - 1);
        t = Clamp(t, 0, l.vSize() - 1);
        break;
    case TEXTURE_BLACK: {
        static const T black = 0.f;
        if (s < 0 || s >= (int)l.uSize() ||
            t < 0 || t >= (int)l.vSize())
            return black;
        break;
    }
}
```

BlockedArray 1017
BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
MIPMap 530
MIPMap::pyramid 535
MIPMap::Texel() 535
TEXTURE_BLACK 530
TEXTURE_CLAMP 530
TEXTURE_REPEAT 530

For nonsquare images, the resolution in one direction must be clamped to one for the upper levels of the image pyramid, where there is still downsampling to do in the larger of the two resolutions. This is handled by the following `max()` calls:

⟨*Initialize i th MIPMap level from i − 1st level*⟩ ≡ 534
```
uint32_t sRes = max(1u, pyramid[i-1]->uSize()/2);
uint32_t tRes = max(1u, pyramid[i-1]->vSize()/2);
pyramid[i] = new BlockedArray<T>(sRes, tRes);
⟨Filter four texels from finer level of pyramid 536⟩
```

The `MIPMap` uses a simple box filter to average four texels from the previous level to find the value at the current texel. The Lanczos filter here would give a slightly better result for this computation, although this modification is left as an exercise at the end of the chapter.

⟨*Filter four texels from finer level of pyramid*⟩ ≡ 536
```
for (uint32_t t = 0; t < tRes; ++t)
    for (uint32_t s = 0; s < sRes; ++s)
        (*pyramid[i])(s, t) = .25f *
           (Texel(i-1, 2*s, 2*t)   + Texel(i-1, 2*s+1, 2*t) +
            Texel(i-1, 2*s, 2*t+1) + Texel(i-1, 2*s+1, 2*t+1));
```

10.4.3 ISOTROPIC TRIANGLE FILTER

The first of the two `MIPMap::Lookup()` methods uses a triangle filter over the texture samples to remove high frequencies. Although this filter function does not give high-quality results, it can be implemented very efficiently. In addition to the (s, t) coordinates of the evaluation point, the caller passes this method a filter width for the lookup, giving the extent of the region of the texture to filter across. This method filters over a square region in texture space, so the width should be conservatively chosen to avoid aliasing in both the s and t directions. Filtering techniques like this one that do not support a filter extent that is nonsquare or non-axis-aligned are known as *isotropic*. The primary disadvantage of isotropic filtering algorithms is that textures viewed at an oblique angle will appear blurry, since the required sampling rate along one axis will be very different from the sampling rate along the other in this case.

Because filtering over many texels for wide filter widths would be inefficient, this method chooses a MIP map level from the pyramid such that the filter region at that level would cover four texels at that level. Figure 10.13 illustrates this idea.

⟨*MIPMap Method Definitions*⟩ +≡
```
template <typename T>
T MIPMap<T>::Lookup(float s, float t, float width) const {
    ⟨Compute MIPMap level for trilinear filtering 537⟩
    ⟨Perform trilinear interpolation at appropriate MIPMap level 537⟩
}
```

BlockedArray 1017
BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
MIPMap 530
MIPMap::Lookup() 540
MIPMap::Texel() 535

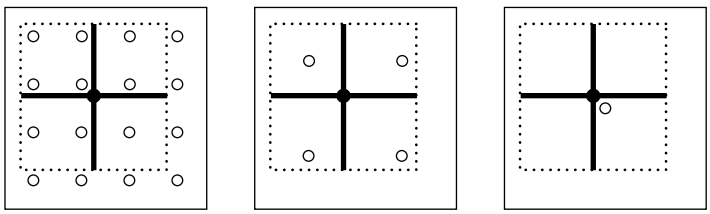


Figure 10.13: Choosing a MIP Map Level for the Triangle Filter. The `MIPMap` chooses a level such that the filter covers four texels.

Since the resolutions of the levels of the pyramid are all powers of two, the resolution of level l is $2^{\text{nLevels}-1-l}$. Therefore, to find the level with a texel spacing width w requires solving

$$\frac{1}{w} = 2^{\text{nLevels}-1-l}$$

for l. In general, this will be a floating-point value between two MIP map levels.

⟨*Compute MIPMap level for trilinear filtering*⟩ ≡ 536
```
float level = nLevels - 1 + Log2(max(width, 1e-8f));
```

As shown by Figure 10.13, applying a triangle filter to the four texels around the sample point will either filter over too small a region or too large a region (except for very carefully selected filter widths). The implementation here applies the triangle filter at both of these levels and blends between them according to how close `level` is to each of them. This helps hide the transitions from one MIP map level to the next at nearby pixels in the final image. While applying a triangle filter to four texels at two levels in this manner doesn't give exactly the same result as applying it to the original highest-resolution texels, the difference isn't too bad in practice and the efficiency of this approach is worth this penalty. In any case, the elliptically weighted average filtering in the next section should be used when texture quality is important.

⟨*Perform trilinear interpolation at appropriate MIPMap level*⟩ ≡ 536
```
if (level < 0)
    return triangle(0, s, t);
else if (level >= nLevels - 1)
    return Texel(nLevels-1, 0, 0);
else {
    uint32_t iLevel = Floor2Int(level);
    float delta = level - iLevel;
    return (1.f-delta) * triangle(iLevel, s, t) +
           delta * triangle(iLevel+1, s, t);
}
```

Floor2Int() 1002
Log2() 1001
MIPMap::nLevels 535
MIPMap::Texel() 535
MIPMap::triangle() 539

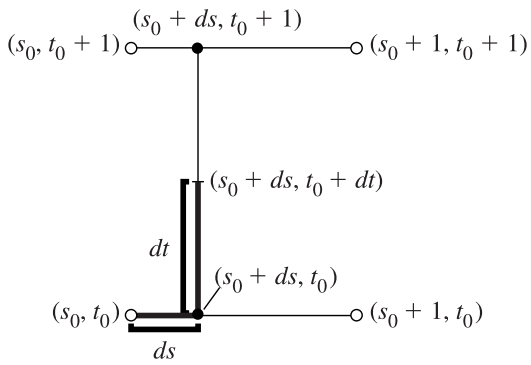


Figure 10.14: To compute the value of the image texture function at an arbitrary (s, t) position, `MIPMap::triangle()` finds the four texels around (s, t) and weights them according to a triangle filter based on their distance to (s, t). One way to implement this is as a series of linear interpolations, as shown here: First, the two texels below (s, t) are linearly interpolated to find a value at $(s, 0)$, and the two texels above it are interpolated to find $(s, 1)$. Then, $(s, 0)$ and $(s, 1)$ are linearly interpolated again to find the value at (s, t).

Given floating-point texture coordinates in $[0, 1]^2$, the `MIPMap::triangle()` routine uses a triangle filter to interpolate between the four texels that surround the sample point, as shown in Figure 10.14. This method first scales the coordinates by the texture resolution at the given MIP map level in each direction, turning them into continuous texel coordinates. Because these are continuous coordinates, but the texels in the image map are defined at discrete texture coordinates, it's important to carefully convert into a common representation. Here, we will do all of our work in discrete coordinates, mapping the continuous texture coordinates to discrete space.

For example, consider the 1D case with a continuous texture coordinate of 2.4: this coordinate is a distance of 0.1 below the discrete texel coordinate 2 (which corresponds to a continuous coordinate of 2.5), and is 0.9 above the discrete coordinate 1 (continuous coordinate 1.5). Thus, if we subtract 0.5 from the continuous coordinate 2.4, giving 1.9, we can correctly compute the correct distances to the discrete coordinates 1 and 2 by subtracting coordinates.

After computing the distances in s and t to the texel at the lower left of the given coordinates, `ds` and `dt`, `MIPMap::triangle()` determines weights for the four texels and computes the filtered value. Recall that the triangle filter is

$$f(x, y) = (1 - |x|)(1 - |y|);$$

BilerpTexture::
Evaluate() 523

MIPMap::triangle() 539

the appropriate weights follow directly. Notice the similarity between this computation and `BilerpTexture::Evaluate()`.

⟨*MIPMap Method Definitions*⟩ +≡

```
template <typename T>
T MIPMap<T>::triangle(uint32_t level, float s, float t) const {
    level = Clamp(level, 0, nLevels-1);
    s = s * pyramid[level]->uSize() - 0.5f;
    t = t * pyramid[level]->vSize() - 0.5f;
    int s0 = Floor2Int(s), t0 = Floor2Int(t);
    float ds = s - s0, dt = t - t0;
    return (1.f-ds) * (1.f-dt) * Texel(level, s0, t0) +
           (1.f-ds) * dt       * Texel(level, s0, t0+1) +
           ds       * (1.f-dt) * Texel(level, s0+1, t0) +
           ds       * dt       * Texel(level, s0+1, t0+1);
}
```

⋆ 10.4.4 ELLIPTICALLY WEIGHTED AVERAGE

The elliptically weighted average (EWA) algorithm fits an ellipse to the two axes in texture space given by the texture coordinate differentials and then filters the texture with a Gaussian filter function (Figure 10.15). It is widely regarded as one of the best texture filtering algorithms in graphics and has been carefully derived from the basic principles of sampling theory. Unlike the triangle filter in the previous section, it can filter over arbitrarily oriented regions of the texture, with some flexibility of having different filter extents in different directions. This type of filter is known as *anisotropic*. This capability greatly improves the quality of its results, since it can properly adapt to different sampling rates along the two image axes.

We won't show the full derivation of this filter here, although we do note that it is distinguished by being a *unified resampling filter*: it simultaneously computes the result of a Gaussian filtered texture function convolved with a Gaussian reconstruction filter in image space. This is in contrast to many other texture filtering methods that ignore the effect of the image space filter or equivalently assume that it is a box. Even if a

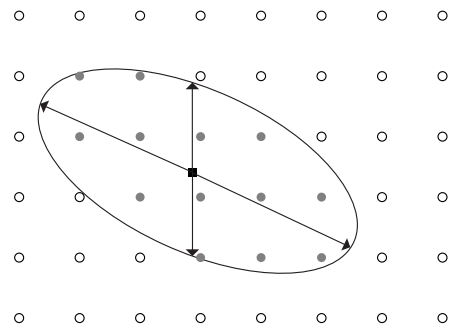


Figure 10.15: The EWA filter applies a Gaussian filter to the texels in an elliptical area around the evaluation point. The extent of the ellipse is such that its edge passes through the positions of the adjacent texture samples as estimated by the texture coordinate partial derivatives.

BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
Clamp() 1000
Floor2Int() 1002
MIPMap 530
MIPMap::nLevels 535
MIPMap::Texel() 535

Gaussian isn't being used for filtering the samples for the image being rendered, taking some account of the spatial variation of the image filter improves the results, assuming that the filter being used is somewhat similar in shape to the Gaussian, as the Mitchell and windowed sinc filters are.

⟨*MIPMap Method Definitions*⟩ +≡
```
template <typename T>
T MIPMap<T>::Lookup(float s, float t, float ds0, float dt0,
                    float ds1, float dt1) const {
    if (doTrilinear) {
        T val = Lookup(s, t,
                       2.f * max(max(fabsf(ds0), fabsf(dt0)),
                                 max(fabsf(ds1), fabsf(dt1))));
        return val;
    }
    ⟨Compute ellipse minor and major axes 540⟩
    ⟨Clamp ellipse eccentricity if too large 540⟩
    ⟨Choose level of detail for EWA lookup and perform EWA filtering 541⟩
}
```

The screen space partial derivatives of the texture coordinates define the axes of the ellipse. The lookup method starts out by determining which of the two axes is the major axis (the longer of the two) and which is the minor, swapping them if needed so that `(ds0,dt0)` is the major axis. The length of the minor axis will be used shortly to select a MIP map level.

⟨*Compute ellipse minor and major axes*⟩ ≡ 540
```
if (ds0*ds0 + dt0*dt0 < ds1*ds1 + dt1*dt1) {
    swap(ds0, ds1);
    swap(dt0, dt1);
}
float majorLength = sqrtf(ds0*ds0 + dt0*dt0);
float minorLength = sqrtf(ds1*ds1 + dt1*dt1);
```

Next the *eccentricity* of the ellipse is computed—the ratio of the length of the major axis to the length of the minor axis. A large eccentricity indicates a very long and skinny ellipse. Because this method filters texels from a MIP map level chosen based on the length of the minor axis, highly eccentric ellipses mean that a large number of texels need to be filtered. To avoid this expense (and to ensure that any EWA lookup takes a bounded amount of time), the length of the minor axis may be increased to limit the eccentricity. The result may be an increase in blurring, although this effect usually isn't noticeable in practice.

⟨*Clamp ellipse eccentricity if too large*⟩ ≡ 540
```
if (minorLength * maxAnisotropy < majorLength && minorLength > 0.f) {
    float scale = majorLength / (minorLength * maxAnisotropy);
    ds1 *= scale;
    dt1 *= scale;
    minorLength *= scale;
}
```

MIPMap 530
MIPMap::doTrilinear 530
MIPMap::maxAnisotropy 530

```
if (minorLength == 0.f) {
    T val = triangle(0, s, t);
    return val;
}
```

Like the triangle filter, the EWA filter uses the image pyramid to reduce the number of texels to be filtered for a particular texture lookup, choosing a MIP map level based on the length of the minor axis. Given the limited eccentricity of the ellipse due to the clamping above, the total number of texels used is thus bounded. Given the length of the minor axis, the computation to find the appropriate pyramid level is the same as was used for the triangle filter. Similarly, the implementation here blends between the filtered results at the two levels around the computed level of detail, again to reduce artifacts from transitions from one level to another.

⟨*Choose level of detail for EWA lookup and perform EWA filtering*⟩ ≡ 540

```
float lod = max(0.f, nLevels - 1.f + Log2(minorLength));
uint32_t ilod = Floor2Int(lod);
float d = lod - ilod;
T val = (1.f - d) * EWA(ilod,   s, t, ds0, dt0, ds1, dt1) +
                d  * EWA(ilod+1, s, t, ds0, dt0, ds1, dt1);
return val;
```

The `MIPMap::EWA()` method actually applies the filter at a particular level.

⟨*MIPMap Method Definitions*⟩ +≡

```
template <typename T>
T MIPMap<T>::EWA(uint32_t level, float s, float t, float ds0, float dt0,
                 float ds1, float dt1) const {
    if (level >= nLevels) return Texel(nLevels-1, 0, 0);
    ⟨Convert EWA coordinates to appropriate scale for level 541⟩
    ⟨Compute ellipse coefficients to bound EWA filter region 542⟩
    ⟨Compute the ellipse's (s, t) bounding box in texture space 542⟩
    ⟨Scan over ellipse bound and compute quadratic equation 543⟩
}
```

This method first converts from texture coordinates in [0, 1] to coordinates and differentials in terms of the resolution of the chosen MIP map level. It also subtracts 0.5 from the continuous position coordinate to align the sample point with the discrete texel coordinates, as was done in `MIPMap::triangle()`.

BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
Floor2Int() 1002
Log2() 1001
MIPMap 530
MIPMap::EWA() 541
MIPMap::nLevels 535
MIPMap::Texel() 535
MIPMap::triangle() 539

⟨*Convert EWA coordinates to appropriate scale for level*⟩ ≡ 541

```
s = s * pyramid[level]->uSize() - 0.5f;
t = t * pyramid[level]->vSize() - 0.5f;
ds0 *= pyramid[level]->uSize();
dt0 *= pyramid[level]->vSize();
ds1 *= pyramid[level]->uSize();
dt1 *= pyramid[level]->vSize();
```

It next computes the coefficients of the implicit equation for the ellipse with axes `(ds0,dt0)` and `(ds1,dt1)` and centered at the origin. Placing the ellipse at the origin

rather than at (s, t) simplifies the implicit equation and the computation of its coefficients, and can be easily corrected for when the equation is evaluated later. The general form of the implicit equation for all points (s, t) inside such an ellipse is

$$e(s, t) = As^2 + Bst + Ct^2 < F,$$

although it is more computationally efficient to divide through by F and express this as

$$e(s, t) = \frac{A}{F}s^2 + \frac{B}{F}st + \frac{C}{F}t^2 = A's^2 + B'st + C't^2 < 1.$$

We will not derive the equations that give the values of the coefficients, although the interested reader can easily verify their correctness.[5]

⟨*Compute ellipse coefficients to bound EWA filter region*⟩ ≡ 541

```
float A = dt0*dt0 + dt1*dt1 + 1;
float B = -2.f * (ds0*dt0 + ds1*dt1);
float C = ds0*ds0 + ds1*ds1 + 1;
float invF = 1.f / (A*C - B*B*0.25f);
A *= invF;
B *= invF;
C *= invF;
```

The next step is to find the axis-aligned bounding box in discrete integer texel coordinates of the texels that are potentially inside the ellipse. The EWA algorithm loops over all of these candidate texels, filtering the contributions of those that are in fact inside the ellipse. The bounding box is found by determining the minimum and maximum values that the ellipse takes in the s and t directions. These extrema can be calculated by finding the partial derivatives $\partial e/\partial s$ and $\partial e/\partial t$, finding their solutions for $s = 0$ and $t = 0$, and adding the offset to the ellipse center. For brevity, we will not include the derivation for these expressions here.

⟨*Compute the ellipse's* (s, t) *bounding box in texture space*⟩ ≡ 541

```
float det = -B*B + 4.f*A*C;
float invDet = 1.f / det;
float uSqrt = sqrtf(det * C), vSqrt = sqrtf(A * det);
int s0 = Ceil2Int (s - 2.f * invDet * uSqrt);
int s1 = Floor2Int(s + 2.f * invDet * uSqrt);
int t0 = Ceil2Int (t - 2.f * invDet * vSqrt);
int t1 = Floor2Int(t + 2.f * invDet * vSqrt);
```

Now that the bounding box is known, the EWA algorithm loops over the texels, transforming each one to the coordinate system where the texture lookup point (s, t) is at the origin with a translation. It then evaluates the ellipse equation to see if the texel is inside the ellipse (Figure 10.16). The value of the implicit ellipse equation $e(s, t)$ gives the squared ratio of the distance from the texel to the ellipse center to the distance from

5 Heckbert's thesis has the original derivation (Heckbert 1989, p. 80). A and C have an extra term of 1 added to them so the ellipse is a minimum of one texel separation wide. This ensures that the ellipse will not fall between the texels when magnifying at the most detailed level.

Ceil2Int() 1002
Floor2Int() 1002

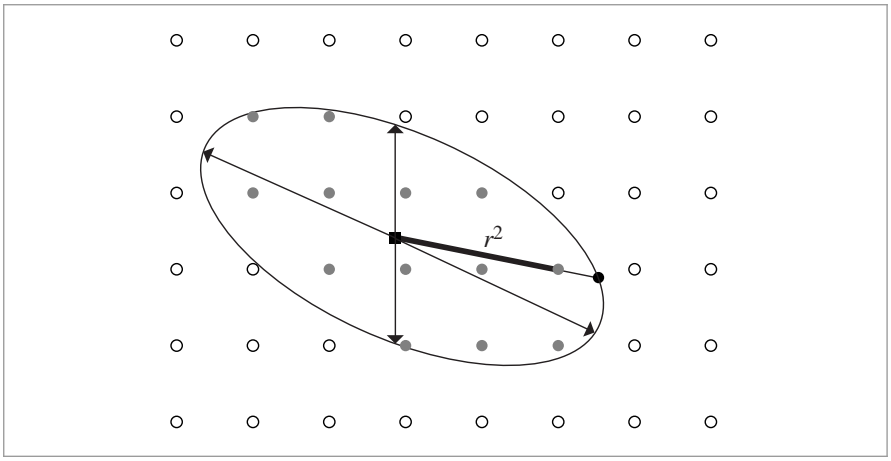


Figure 10.16: Finding the r^2 Ellipse Value for the EWA Filter Table Lookup.

the ellipse edge to the center along the line through the texel. If it is inside, the weight of the texel is computed with a Gaussian centered at the middle of the ellipse. The final filtered value returned is a weighted sum over texels (s', t') inside the ellipse, where f is the Gaussian filter function:

$$\frac{\sum f(s'-s, t'-t)t(s', t')}{\sum f(s'-s, t'-t)}.$$

⟨*Scan over ellipse bound and compute quadratic equation*⟩ ≡ 541

```
T sum(0.);
float sumWts = 0.f;
for (int it = t0; it <= t1; ++it) {
    float tt = it - t;
    for (int is = s0; is <= s1; ++is) {
        float ss = is - s;
        ⟨Compute squared radius and filter texel if inside ellipse 544⟩
    }
}
return sum / sumWts;
```

A nice feature of the implicit equation $e(s, t)$ is that its value at a particular texel is the squared ratio of the distance from the center of the ellipse to the texel to the distance from the center of the ellipse to the ellipse boundary along the line through that texel (Figure 10.16). This value is used to index into a precomputed lookup table of Gaussian filter function values.

⟨*Compute squared radius and filter texel if inside ellipse*⟩ ≡ 543
```
float r2 = A*ss*ss + B*ss*tt + C*tt*tt;
if (r2 < 1.) {
    float weight = weightLut[min(Float2Int(r2 * WEIGHT_LUT_SIZE),
                                 WEIGHT_LUT_SIZE-1)];
    sum += Texel(level, is, it) * weight;
    sumWts += weight;
}
```

The lookup table is initialized the first time a `MIPMap` is constructed. Because it will be indexed with squared distances from the filter center r^2, each entry stores a value $e^{-\alpha r}$, rather than $e^{-\alpha r^2}$.

⟨*MIPMap Private Data*⟩ +≡ 530
```
#define WEIGHT_LUT_SIZE 128
static float *weightLut;
```

So that the filter function goes to zero at the end of its extent rather than having an abrupt step, `expf(-alpha)` is subtracted from the filter values here.

⟨*Initialize EWA filter weights if needed*⟩ ≡ 530
```
if (!weightLut) {
    weightLut = AllocAligned<float>(WEIGHT_LUT_SIZE);
    for (int i = 0; i < WEIGHT_LUT_SIZE; ++i) {
        float alpha = 2;
        float r2 = float(i) / float(WEIGHT_LUT_SIZE - 1);
        weightLut[i] = expf(-alpha * r2) - expf(-alpha);
    }
}
```

10.5 SOLID AND PROCEDURAL TEXTURING

Once one starts to think of the (s, t) texture coordinates used by 2D texture functions as quantities that can be computed by arbitrary functions and not just from the parametric coordinates of the surface, it is natural to generalize texture functions to be defined over three-dimensional domains (often called *solid textures*) rather than just 2D (s, t). One reason solid textures are particularly convenient is that all objects have a natural three-dimensional texture mapping—object space position. This is a substantial advantage for texturing objects that don't have a natural two-dimensional parameterization (e.g., triangle meshes and implicit surfaces), and for objects that have a distorted parameterization (e.g., near the poles of a sphere). In preparation for this idea, Section 10.2.5 defined a general `TextureMapping3D` interface to compute 3D texture coordinates as well as an `IdentityMapping3D` implementation.

Solid textures introduce a new problem, however: texture representation. A three-dimensional image map takes up a fair amount of storage space and is much harder to acquire than a two-dimensional texture map, which can be extracted from photographs

AllocAligned() 1013
Float2Int() 1002
IdentityMapping3D 519
MIPMap 530
MIPMap::Texel() 535
MIPMap::weightLut 544
MIPMap::WEIGHT_LUT_SIZE 544
TextureMapping3D 519

or painted by an artist. Therefore, procedural texturing—the idea that programs could be executed to generate texture values at arbitrary positions on surfaces in the scene—came into use at the same time that solid texturing was developed. A simple example of procedural texturing is a procedural sine wave. If we wanted to use a sine wave for bump mapping (for example, to simulate waves in water), it would be inefficient and potentially inaccurate to precompute values of the function at a grid of points and then store them in an image map. Instead, it makes much more sense to evaluate the `sin()` function at points on the surface as needed.

If we can find a three-dimensional function that describes the colors of the grain in a solid block of wood, for instance, then we can generate images of complex objects that appear to be carved from wood. Over the years, procedural texturing has grown in application considerably as techniques have been developed to describe more and more complex surfaces procedurally.

Procedural texturing has a number of interesting implications. First, it can be used to reduce memory requirements for rendering, by reducing the need for the storage of large, high-resolution texture maps. In addition, procedural shading gives the promise of potentially infinite detail; as the viewer approaches an object, the texturing function is evaluated at the points being shaded, which naturally leads to the right amount of detail being visible. In contrast, image texture maps become blurry when the viewer is too close to them. On the other hand, subtle details of the appearance of procedural textures can be much more difficult to control than image maps.

Another difficulty with procedural textures is antialiasing. Procedural textures are often expensive to evaluate, and sets of point samples that fully characterize their behavior aren't available as they are for with image maps. Because we would like to remove high-frequency information in the texture function before we take samples from it, we need to be aware of the frequency content of the various steps we take along the way so we can avoid introducing high frequencies. Although this sounds daunting, there are a handful of techniques that work well to handle this issue.

10.5.1 UV TEXTURE

Our first procedural texture converts the surface's (u, v) coordinates into the red and green components of a `Spectrum` (Figure 10.17). It is especially useful when debugging the parameterization of a new `Shape`, for example. It is defined in `textures/uv.h` and `textures/uv.cpp`.

```
⟨UVTexture Declarations⟩ ≡
   class UVTexture : public Texture<Spectrum> {
   public:
       ⟨UVTexture Public Methods 546⟩
   private:
       TextureMapping2D *mapping;
   };
```

Shape 108
Spectrum 263
Texture 519
TextureMapping2D 514

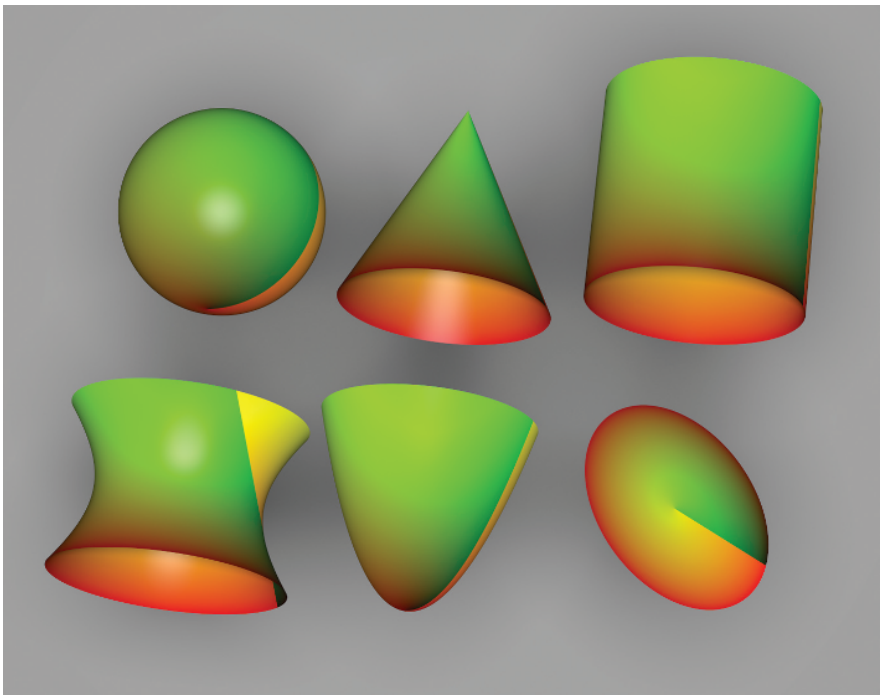


Figure 10.17: The UV Texture Applied to All of pbrt's Quadric Shapes. The u parameter is mapped to the red channel, and the v parameter is mapped to green.

⟨*UVTexture Public Methods*⟩ ≡ 545

```
Spectrum Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    float rgb[3] = { s - Floor2Int(s), t - Floor2Int(t), 0.f };
    return Spectrum::FromRGB(rgb);
}
```

10.5.2 CHECKERBOARD

The checkerboard is the canonical procedural texture (Figure 10.18). The (s, t) texture coordinates are used to break up parameter space into square regions that are shaded with alternating patterns. Rather than just supporting checkerboards that switch between two fixed colors, the implementation here allows the user to pass in two textures to color the alternating regions. The traditional black-and-white checkerboard is obtained by passing two `ConstantTextures`. Its implementation is in the files `textures/checkerboard.h` and `textures/checkerboard.cpp`.

ConstantTexture 520
DifferentialGeometry 102
Floor2Int() 1002
Spectrum 263
Spectrum::FromRGB() 277
TextureMapping2D::Map() 514
UVTexture::mapping 545

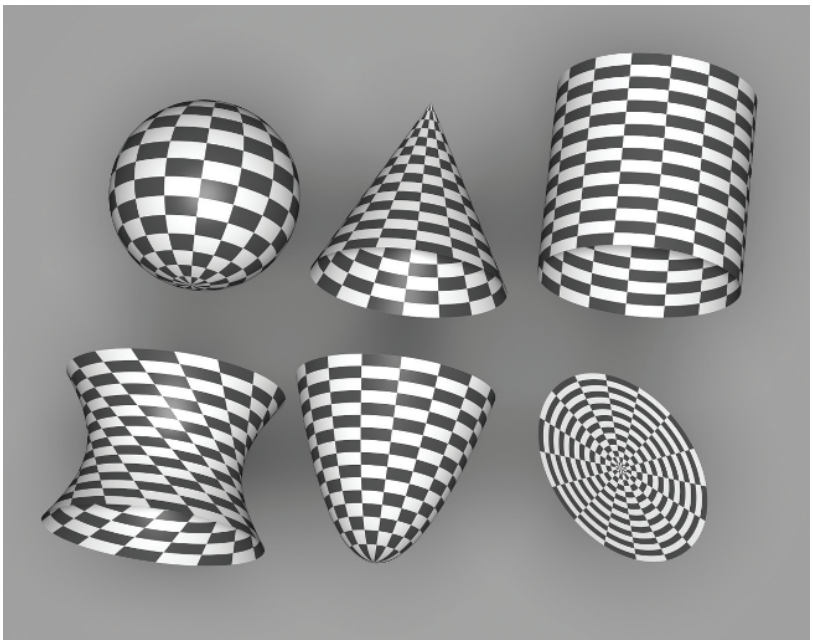


Figure 10.18: The Checkerboard Texture Applied to All of pbrt's Quadric Shapes.

⟨*CheckerboardTexture Declarations*⟩ ≡
```
template <typename T> class Checkerboard2DTexture : public Texture<T> {
public:
    ⟨Checkerboard2DTexture Public Methods 547⟩
private:
    ⟨Checkerboard2DTexture Private Data 547⟩
};
```

For simplicity, the frequency of the check function is 1 in (s, t) space: checks are one unit wide in each direction. The effective frequency can always be changed by the `TextureMapping2D` class with an appropriate scale of the (s, t) coordinates.

⟨*Checkerboard2DTexture Public Methods*⟩ ≡ 547
```
Checkerboard2DTexture(TextureMapping2D *m, Reference<Texture<T> > c1,
                      Reference<Texture<T> > c2, const string &aa)
    : mapping(m), tex1(c1), tex2(c2) {
    ⟨Select antialiasing method for Checkerboard2DTexture 548⟩
}
```

⟨*Checkerboard2DTexture Private Data*⟩ ≡ 547
```
TextureMapping2D *mapping;
Reference<Texture<T> > tex1, tex2;
```

Checkerboard2DTexture 547
Checkerboard2DTexture::mapping 547
Checkerboard2DTexture::tex1 547
Checkerboard2DTexture::tex2 547
Reference 1011
Texture 519
TextureMapping2D 514

The checkerboard is good for demonstrating trade-offs between various antialiasing approaches for procedural textures. The implementation here supports both simple point sampling (no antialiasing) and a closed-form box filter evaluated over the filter region. The image sequence in Figure 10.22 at the end of this section shows the results of these approaches.

⟨*Select antialiasing method for* Checkerboard2DTexture⟩ ≡ 547
```
if (aa == "none")             aaMethod = NONE;
else if (aa == "closedform")  aaMethod = CLOSEDFORM;
else {
    Warning("Antialiasing mode \"%s\" not understood by "
            "Checkerboard2DTexture; using \"closedform\"", aa.c_str());
    aaMethod = CLOSEDFORM;
}
```

⟨*Checkerboard2DTexture Private Data*⟩ +≡ 547
```
enum { NONE, CLOSEDFORM } aaMethod;
```

The evaluation routine does the usual texture coordinate and differential computation and then uses the appropriate fragment to compute an antialiased checkerboard value (or not antialiased, if point sampling has been selected).

⟨*Checkerboard2DTexture Public Methods*⟩ +≡ 547
```
T Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    if (aaMethod == NONE) {
        ⟨Point sample Checkerboard2DTexture 548⟩
    }
    else {
        ⟨Compute closed-form box-filtered Checkerboard2DTexture value 549⟩
    }
}
```

The simplest case is to ignore antialiasing and just point-sample the checkerboard texture at the point. For this case, after getting the (s, t) texture coordinates from the TextureMapping2D, the integer checkerboard coordinates for that (s, t) position are computed, added together, and checked for odd or even parity to determine which of the two textures to evaluate.

⟨*Point sample* Checkerboard2DTexture⟩ ≡ 548, 550
```
if ((Floor2Int(s) + Floor2Int(t)) % 2 == 0)
    return tex1->Evaluate(dg);
return tex2->Evaluate(dg);
```

Given how bad aliasing can be in a point-sampled checkerboard texture, we will invest some effort to antialias it properly. The easiest case happens when the entire filter region lies inside a single check (Figure 10.19). In this case, we simply need to determine which of the check types we are inside and evaluate that one. As long as the Texture inside that check does appropriate antialiasing itself, the result for this case will be antialiased.

Checkerboard2DTexture::
aaMethod 548
Checkerboard2DTexture::
CLOSEDFORM 548
Checkerboard2DTexture::
mapping 547
Checkerboard2DTexture::
NONE 548
Checkerboard2DTexture::
tex1 547
Checkerboard2DTexture::
tex2 547
DifferentialGeometry 102
Floor2Int() 1002
Texture 519
Texture::Evaluate() 520
TextureMapping2D 514
TextureMapping2D::Map() 514
Warning() 1005

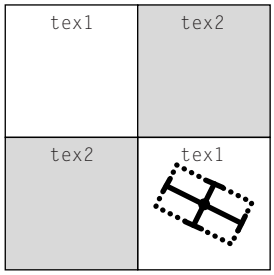


Figure 10.19: The Easy Case for Filtering the Checkerboard. If the filter region around the lookup point is entirely in one check, the checkerboard texture doesn't need to worry about antialiasing and can just evaluate the texture for that check.

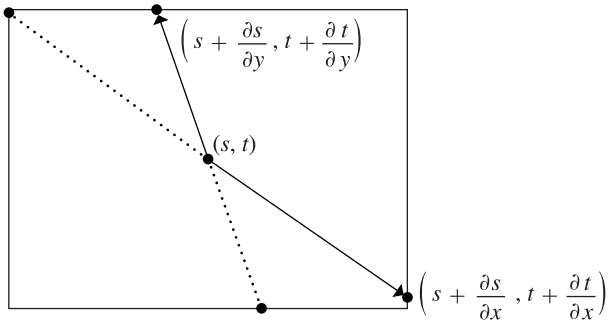


Figure 10.20: It is often convenient to use the axis-aligned bounding box around the texture evaluation point and the offsets from its partial derivatives as the region to filter over. Here, it's easy to see that the lengths of sides of the box are $2\max(|\partial s/\partial x|, |\partial s/\partial y|)$ and $2\max(|\partial t/\partial x|, |\partial t/\partial y|)$.

⟨*Compute closed-form box-filtered* `Checkerboard2DTexture` *value*⟩ ≡ 548
⟨*Evaluate single check if filter is entirely inside one of them* 550⟩
⟨*Apply box filter to checkerboard region* 551⟩

It's straightforward to check if the entire filter region is inside a single check by computing its bounding box and seeing if its extent lies inside the same check. For the remainder of this section, we will use the axis-aligned bounding box of the filter region given by the partial derivatives $\partial s/\partial x$, $\partial s/\partial y$, and so on, as the area to filter over, rather than trying to filter over the ellipse defined by the partial derivatives as the EWA filter did (Figure 10.20). This simplifies the implementation here, although somewhat increases the blurriness of the filtered values. The variables `ds` and `dt` in the following hold half the filter width in each direction, so the total area filtered over ranges from `(s-ds, t-dt)` to `(s+ds, t+dt)`.

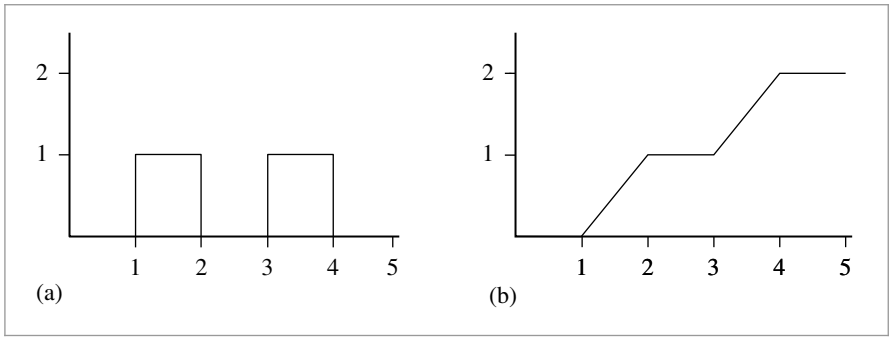


Figure 10.21: Integrating the Checkerboard Step Function. (a) The 1D step function that defines the checkerboard texture function, $c(x)$. (b) A graph of the value of the integral $\int_0^x c(x)\mathrm{d}x$.

⟨*Evaluate single check if filter is entirely inside one of them*⟩ ≡ 549

```
float ds = max(fabsf(dsdx), fabsf(dsdy));
float dt = max(fabsf(dtdx), fabsf(dtdy));
float s0 = s - ds, s1 = s + ds;
float t0 = t - dt, t1 = t + dt;
if (Floor2Int(s0) == Floor2Int(s1) && Floor2Int(t0) == Floor2Int(t1)) {
    ⟨Point sample Checkerboard2DTexture 548⟩
}
```

Otherwise, the lookup method approximates the filtered value by first computing a floating-point value that indicates what fraction of the filter region covers each of the two check types. This is equivalent to computing the average of the 2D step function that takes on the value 0 when we are in `tex1` and 1 when we are in `tex2`, over the filter region. Figure 10.21(a) shows a graph of the checkerboard function $c(x)$, defined as

$$c(x) = \begin{cases} 0 & \lfloor x \rfloor \text{ is odd} \\ 1 & \text{otherwise.} \end{cases}$$

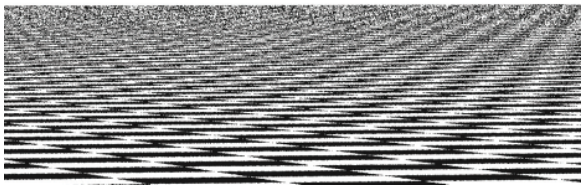
Given the average value, we can blend between the two subtextures, according to what fraction of the filter region each one is visible for.

The integral of the 1D checkerboard function $c(x)$ can be used to compute the average value of the function over some extent. Inspection of the graph reveals that

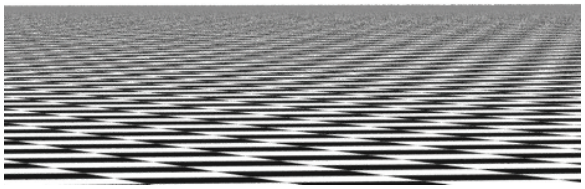
$$\int_0^x c(x)\,\mathrm{d}x = \lfloor x/2 \rfloor + 2\max(x/2 - \lfloor x/2 \rfloor - .5, 0).$$

To compute the average value of the step function in two dimensions, we separately compute the integral of the checkerboard in each 1D direction in order to compute its average value over the filter region.

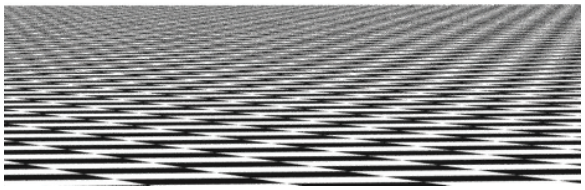
`Floor2Int()` 1002



(a)



(b)



(c)

Figure 10.22: Comparisons of three approaches for antialiasing procedural textures, applied to the checkerboard texture. (a) No effort has been made to remove high-frequency variation from the texture function, so there are severe artifacts in the image, rendered with one sample per pixel. (b) This image shows the approach based on computing the filter region in texture space and averaging the texture function over that area, also rendered with one sample per pixel. (c) Here the checkerboard function was effectively supersampled by taking 16 samples per pixel and then point-sampling the texture. Both the area-averaging and the supersampling approaches give substantially better results than the first approach. In this example, supersampling gives the best results, since the averaging approach has blurred out the checkerboard pattern sooner than was needed because it approximates the filter region with its axis-aligned box.

⟨*Apply box filter to checkerboard region*⟩ ≡ 549

```
#define BUMPINT(x) \
    (Floor2Int((x)/2) + \
     2.f * max((x/2)-Floor2Int(x/2) - .5f, 0.f))
float sint = (BUMPINT(s1) - BUMPINT(s0)) / (2.f * ds);
float tint = (BUMPINT(t1) - BUMPINT(t0)) / (2.f * dt);
float area2 = sint + tint - 2.f * sint * tint;
if (ds > 1.f || dt > 1.f)
    area2 = .5f;
return (1.f - area2) * tex1->Evaluate(dg) +
       area2         * tex2->Evaluate(dg);
```

Checkerboard2DTexture::
tex1 547

Texture::Evaluate() 520



Figure 10.23: The Dragon Model, Textured with the `Checkerboard3DTexture` Procedural Texture. Notice how the model appears to be carved out of 3D checks, rather than having them pasted on its surface.

10.5.3 SOLID CHECKERBOARD

The `Checkerboard2DTexture` class from the previous section wraps a checkerboard pattern *around* the object in parameter space. We can also define a solid checkerboard pattern based on three-dimensional texture coordinates so that the object appears carved out of 3D checker cubes (Figure 10.23). Like the 2D variant, this implementation chooses between texture functions based on the lookup position. Note that these two textures need not be solid textures themselves; the `Checkerboard3DTexture` merely chooses between them based on the 3D position of the point.

⟨*CheckerboardTexture Declarations*⟩ +≡

```
template <typename T> class Checkerboard3DTexture : public Texture<T> {
public:
    ⟨Checkerboard3DTexture Public Methods 552⟩
private:
    ⟨Checkerboard3DTexture Private Data 553⟩
};
```

⟨*Checkerboard3DTexture Public Methods*⟩ ≡ 552

```
Checkerboard3DTexture(TextureMapping3D *m, Reference<Texture<T> > c1,
                      Reference<Texture<T> > c2)
    : mapping(m), tex1(c1), tex2(c2) {
}
```

Checkerboard2DTexture 547
Checkerboard3DTexture 552
Checkerboard3DTexture::mapping 553
Checkerboard3DTexture::tex1 553
Checkerboard3DTexture::tex2 553
Reference 1011
Texture 519
TextureMapping3D 519

⟨*Checkerboard3DTexture Private Data*⟩ ≡ 552
```
TextureMapping3D *mapping;
Reference<Texture<T> > tex1, tex2;
```

Ignoring antialiasing, the basic computation to see if a point is inside a 3D checker region is

```
((Floor2Int(P.x) + Floor2Int(P.y) + Floor2Int(P.z)) % 2 == 0).
```

The `Checkerboard3DTexture` doesn't have any built-in support for antialiasing.

⟨*Checkerboard3DTexture Public Methods*⟩ +≡ 552
```
T Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point p = mapping->Map(dg, &dpdx, &dpdy);
    if ((Floor2Int(p.x) + Floor2Int(p.y) + Floor2Int(p.z)) % 2 == 0)
        return tex1->Evaluate(dg);
    else
        return tex2->Evaluate(dg);
}
```

10.6 NOISE

In order to write solid textures that describe complex surface appearances, it is helpful to be able to introduce some controlled variation to the process. Consider a wood floor made of individual planks; each plank's color is likely to be slightly different than the others. Or consider a windswept lake; we might want to have waves of similar amplitude across the entire lake, but we don't want them to be homogeneous over all parts of the lake (as they might be if they were constructed from a sum of sine waves, for example). Modeling this sort of variation in a texture helps make the final result look more realistic.

One difficulty in developing textures like these is that the renderer evaluates the surface's texture functions at an irregularly distributed set of points, where each evaluation is completely independent of the others. As such, procedural textures must *implicitly* define a complex pattern by answering queries about what the pattern's value is at all of these points. In contrast, the *explicit* pattern description approach is embodied by the PostScript® language, for example, which describes graphics on a page with a series of drawing commands. One difficulty that the implicit approach introduces is that the texture can't just call `RNG::RandomFloat()` at each point at which it is evaluated to introduce randomness: because each point would have a completely different random value than its neighbors, no coherence would be possible in the generated pattern.

An elegant way to address this issue of introducing controlled randomness to procedural textures in graphics is the application of what is known as a *noise function*. In general, noise functions used in graphics are smoothly varying functions taking $\mathbb{R}^n \rightarrow [-1, 1]$, for at least $n = 1, 2, 3$, without obvious repetition. One of the most crucial properties of a practical noise function is that it be band limited with a known maximum frequency. This makes it possible to control the frequency content added to a texture due to the

Checkerboard3DTexture::mapping 553
Checkerboard3DTexture::tex1 553
Checkerboard3DTexture::tex2 553
DifferentialGeometry 102
Floor2Int() 1002
Point 63
Reference 1011
RNG::RandomFloat() 1003
Texture 519
Texture::Evaluate() 520
TextureMapping3D 519
TextureMapping3D::Map() 519
Vector 57

noise function so that frequencies higher than those allowed by the Nyquist limit aren't introduced.

Many of the noise functions that have been developed are built on the idea of an integer lattice over $\mathbb{R}^3$. First, a value is associated with each integer (x, y, z) position in space. Then, given an arbitrary position in space, the eight surrounding lattice values are found. These lattice values are then interpolated to compute the noise value at the particular point. This idea can be generalized or restricted to more or fewer dimensions d, where the number of lattice points is 2^d. A simple example of this approach is *value noise*, where pseudo-random numbers between -1 and 1 are associated with each lattice point, and actual noise values are computed with trilinear interpolation or with a more complex spline interpolant, which can give a smoother result by avoiding derivative discontinuities when moving from one lattice cell to another.

For such a noise function, given an integer (x, y, z) lattice point, it must be possible to efficiently compute its parameter value in a way that always associates the same value with each lattice point. Because it is infeasible to store values for all possible (x, y, z) points, some compact representation is needed. One option is to use a hash function, where the coordinates are hashed and then used to look up parameters from a fixed-size table of precomputed pseudo-random parameter values.

10.6.1 PERLIN NOISE

In `pbrt` we will implement a noise function introduced by Ken Perlin (1985a, 2002); as such, it is known as *Perlin noise*. It has a value of zero at all (x, y, z) integer lattice points. Its variation comes from gradient vectors at each lattice point that guide the interpolation of a smooth function in between the points (Figure 10.24). This noise function has many

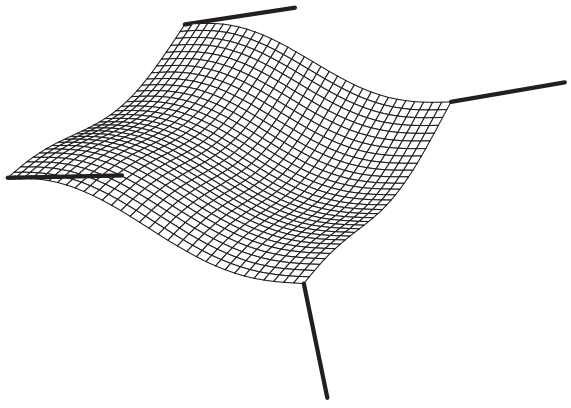


Figure 10.24: The Perlin noise function is computed by generating a smooth function that is zero but with a given derivative at integer lattice points. The derivatives are used to compute a smooth interpolating surface. Here, a 2D slice of the noise function is shown with four gradient vectors.

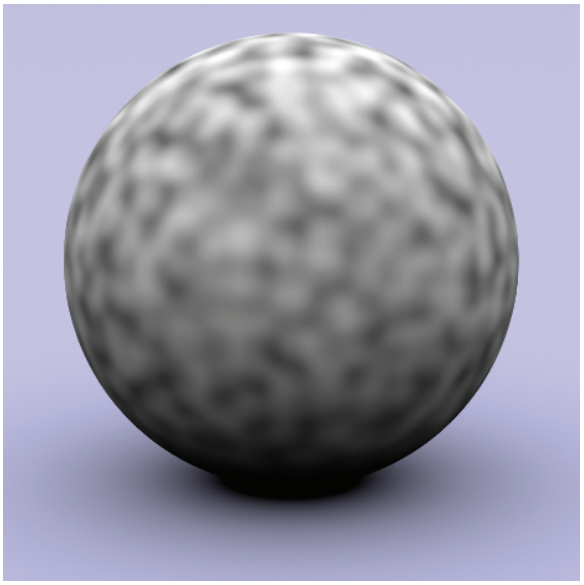


Figure 10.25: Perlin's Noise Function Modulating the Diffuse Color of a Sphere.

of the desired characteristics of a noise function described above, is computationally efficient, and is easy to implement. Figure 10.25 shows its value rendered on a sphere.

⟨*Texture Method Definitions*⟩ +≡
```
float Noise(float x, float y, float z) {
    ⟨Compute noise cell coordinates and offsets 555⟩
    ⟨Compute gradient weights 556⟩
    ⟨Compute trilinear interpolation of weights 558⟩
}
```

For convenience, there is also a variant of `Noise()` that takes a `Point` directly:

⟨*Texture Method Definitions*⟩ +≡
```
float Noise(const Point &P) { return Noise(P.x, P.y, P.z); }
```

The implementation first computes the integer coordinates of the cell that contains the given point and the fractional offsets of the point from the lower cell corner:

⟨*Compute noise cell coordinates and offsets*⟩ ≡ 555
```
int ix = Floor2Int(x), iy = Floor2Int(y), iz = Floor2Int(z);
float dx = x - ix, dy = y - iy, dz = z - iz;
```

It next computes eight weight values, one for each corner of the cell that the point lies inside. Each integer lattice point has a gradient vector associated with it. The influence of the gradient vector for any point inside the cell is obtained by computing the dot product of the vector from the gradient's corner to the lookup point and the gradient

`Floor2Int()` 1002
`Noise()` 555
`Point` 63

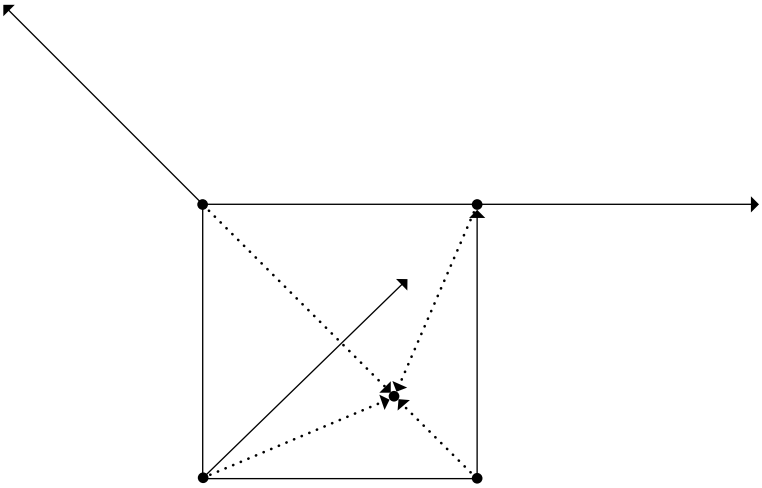


Figure 10.26: The dot product of the vector from the corners of the cell to the lookup point (dotted lines) with each of the gradient vectors (solid lines) gives the influence of each gradient to the noise value at the point.

vector (Figure 10.26); this is handled by the `Grad()` function. Note that the vectors to the corners other than the lower-left one can be easily computed incrementally based on that vector.

⟨*Compute gradient weights*⟩ ≡ 555

```
ix &= (NOISE_PERM_SIZE-1);
iy &= (NOISE_PERM_SIZE-1);
iz &= (NOISE_PERM_SIZE-1);
float w000 = Grad(ix,   iy,   iz,   dx,   dy,   dz);
float w100 = Grad(ix+1, iy,   iz,   dx-1, dy,   dz);
float w010 = Grad(ix,   iy+1, iz,   dx,   dy-1, dz);
float w110 = Grad(ix+1, iy+1, iz,   dx-1, dy-1, dz);
float w001 = Grad(ix,   iy,   iz+1, dx,   dy,   dz-1);
float w101 = Grad(ix+1, iy,   iz+1, dx-1, dy,   dz-1);
float w011 = Grad(ix,   iy+1, iz+1, dx,   dy-1, dz-1);
float w111 = Grad(ix+1, iy+1, iz+1, dx-1, dy-1, dz-1);
```

The gradient vector for a particular integer lattice point is found by indexing into a precomputed table of integer values, `NoisePerm`. The four low-order bits of the table's value for the lattice point determine which of 16 gradient vectors is associated with it. In a preprocessing step, this table of size `NOISE_PERM_SIZE` was filled with numbers from 0 to `NOISE_PERM_SIZE-1` and then randomly permuted. These values were then duplicated, making an array of size `2*NOISE_PERM_SIZE` that holds the table twice in succession. The second copy of the table makes lookups in the following code slightly more efficient.

`Grad()` 557
`NOISE_PERM_SIZE` 557

Given a particular (ix,iy,iz) lattice point, a series of table lookups gives a value from the random-number table:

```
NoisePerm[NoisePerm[NoisePerm[ix]+iy]+iz];
```

By doing three nested permutations in this way, rather than NoisePerm[ix+iy+iz], for example, the final result is more irregular. For example, the first approach usually doesn't generally return the same value if ix and iy are interchanged, as the second does. Furthermore, since the table was replicated to be twice the original length, the lookups can be done as described above, eliminating the need for modulus operations in code along the lines of

```
(NoisePerm[ix]+iy) % NOISE_PERM_SIZE
```

Given a final value from the permutation table that determines the gradient number, the dot product with the corresponding gradient vector must be computed. However, the gradient vectors do not need to be represented explicitly. All of the gradients use only −1, 0, or 1 in their coordinates, so that the dot products reduce to addition of some (possibly negated) components of the vector.[6] The final implementation is the following:

⟨*Texture Method Definitions*⟩ +≡

```
inline float Grad(int x, int y, int z, float dx, float dy, float dz) {
    int h = NoisePerm[NoisePerm[NoisePerm[x]+y]+z];
    h &= 15;
    float u = h<8 || h==12 || h==13 ? dx : dy;
    float v = h<4 || h==12 || h==13 ? dy : dz;
    return ((h&1) ? -u : u) + ((h&2) ? -v : v);
}
```

⟨*Perlin Noise Data*⟩ ≡

```
#define NOISE_PERM_SIZE 256
static int NoisePerm[2 * NOISE_PERM_SIZE] = {
    151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96,
    53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142,
    ⟨Remainder of the noise permutation table⟩
};
```

Given these eight contributions from the gradients, the next step is to trilinearly interpolate between them at the point. Rather than interpolating with dx, dy, and dz directly, though, each of these values is passed through a smoothing function. This ensures that the noise function has first- and second-derivative continuity as lookup points move between lattice cells.

NoisePerm 557

NOISE_PERM_SIZE 557

6 The original formulation of Perlin noise also had a precomputed table of pseudo-random gradient directions, although Perlin has more recently suggested that the randomness from the permutation table is enough to remove regularity from the noise function.

⟨*Texture Method Definitions*⟩ +≡

```
inline float NoiseWeight(float t) {
    float t3 = t*t*t;
    float t4 = t3*t;
    return 6.f*t4*t - 15.f*t4 + 10.f*t3;
}
```

⟨*Compute trilinear interpolation of weights*⟩ ≡ 555

```
float wx = NoiseWeight(dx), wy = NoiseWeight(dy), wz = NoiseWeight(dz);
float x00 = Lerp(wx, w000, w100);
float x10 = Lerp(wx, w010, w110);
float x01 = Lerp(wx, w001, w101);
float x11 = Lerp(wx, w011, w111);
float y0 = Lerp(wy, x00, x10);
float y1 = Lerp(wy, x01, x11);
return Lerp(wz, y0, y1);
```

10.6.2 RANDOM POLKA DOTS

A basic use of the noise function is as part of a polka dot texture that divides (s, t) texture space into rectangular cells (Figure 10.27). Each cell has a 50% chance of having a dot inside of it, and the dots are randomly placed inside their cells. `DotsTexture` takes the

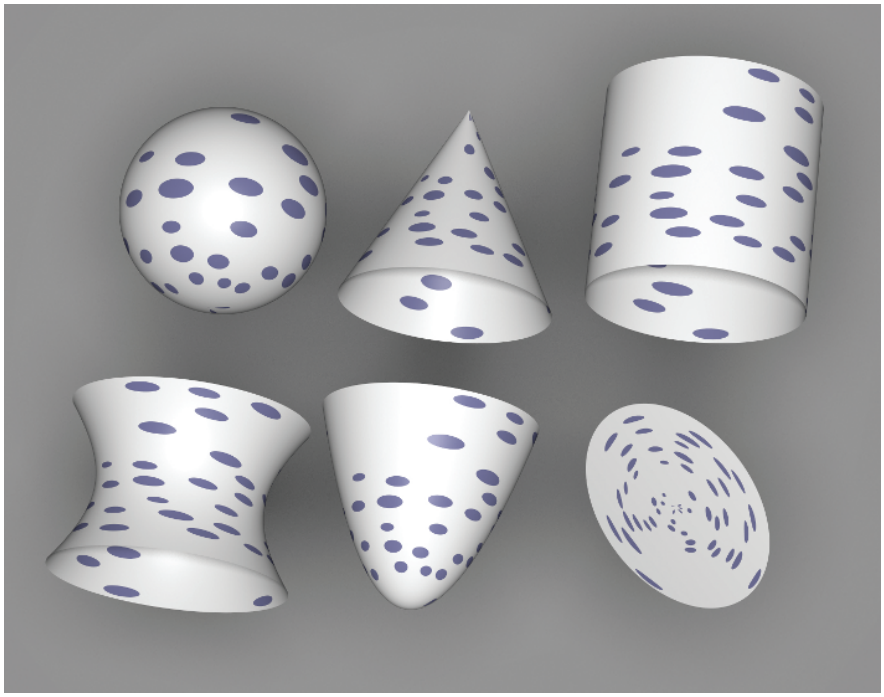


Figure 10.27: The Polka Dot Texture Applied to All of `pbrt`'s Quadric Shapes.

DotsTexture 559
Lerp() 1000
NoiseWeight() 558

usual 2D mapping function, as well as two Textures, one for the regions of the surface outside of the dots and one for the regions inside. It is defined in the files textures/dots.h and textures/dots.cpp.

⟨*DotsTexture Declarations*⟩ ≡
```
template <typename T> class DotsTexture : public Texture<T> {
public:
    ⟨DotsTexture Public Methods 559⟩
private:
    ⟨DotsTexture Private Data 559⟩
};
```

⟨*DotsTexture Public Methods*⟩ ≡ 559
```
DotsTexture(TextureMapping2D *m, Reference<Texture<T> > t1,
            Reference<Texture<T> > t2)
    : mapping(m), outsideDot(t1), insideDot(t2) {
}
```

⟨*DotsTexture Private Data*⟩ ≡ 559
```
TextureMapping2D *mapping;
Reference<Texture<T> > outsideDot, insideDot;
```

The evaluation function starts by taking the (s, t) texture coordinates and computing integer sCell and tCell values, which give the coordinates of the cell they are inside. We will not consider antialiasing of the polka dots texture here; an exercise at the end of the chapter outlines how this might be done.

⟨*DotsTexture Public Methods*⟩ +≡ 559
```
T Evaluate(const DifferentialGeometry &dg) const {
    ⟨Compute cell indices for dots 559⟩
    ⟨Return insideDot result if point is inside dot 560⟩
    return outsideDot->Evaluate(dg);
}
```

⟨*Compute cell indices for dots*⟩ ≡ 559
```
float s, t, dsdx, dtdx, dsdy, dtdy;
mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
int sCell = Floor2Int(s + .5f), tCell = Floor2Int(t + .5f);
```

DifferentialGeometry 102
DotsTexture 559
DotsTexture::insideDot 559
DotsTexture::mapping 559
DotsTexture::outsideDot 559
Floor2Int() 1002
Reference 1011
Texture 519
Texture::Evaluate() 520
TextureMapping2D 514
TextureMapping2D::Map() 514

Once the cell coordinate is known, it's necessary to decide if there is a polka dot in the cell. Obviously, this computation needs to be consistent so that for each time this routine runs for points in a particular cell it returns the same result. Yet we'd like the result not to be completely regular (e.g., with a dot in every other cell). Noise solves this problem: by evaluating the noise function at a position that is the same for all points inside this cell—sCell+.5, tCell+.5—we can compute an irregularly varying but consistent value for each cell.[7] If this value is greater than zero, a dot is placed in the cell.

7 Recall that the noise function always returns zero at integer (x, y, z) coordinates, so we don't want to just evaluate it at (sCell, tCell). Although the 3D noise function would actually be evaluating noise at sCell, tCell, .5, slices through noise with integer values for any of the coordinates are not as well-distributed as with all of them offset.

If there is a dot in the cell, the noise function is used again to randomly shift the center of the dot within the cell. The points at which the noise function is evaluated for the center shift are offset by arbitrary constant amounts, however, so that noise values from different noise cells are used from them, eliminating a possible source of correlation with the noise value used to determine the presence of a dot in the first place. (But the dot's radius needs to be small enough so that it doesn't spill over the cell's boundary after being shifted; in that case, points where the texture was being evaluated would also need to consider the dots based in neighboring cells as potentially affecting their value.)

Given the dot center and radius, the texture needs to decide if the (s, t) coordinates are within the radius of the shifted center. It does this by computing their squared distance to the center and comparing it to the squared radius.

⟨*Return* `insideDot` *result if point is inside dot*⟩ ≡ 559

```
if (Noise(sCell+.5f, tCell+.5f) > 0) {
    float radius = .35f;
    float maxShift = 0.5f - radius;
    float sCenter = sCell + maxShift *
        Noise(sCell + 1.5f, tCell + 2.8f);
    float tCenter = tCell + maxShift *
        Noise(sCell + 4.5f, tCell + 9.8f);
    float ds = s - sCenter, dt = t - tCenter;
    if (ds*ds + dt*dt < radius*radius)
        return insideDot->Evaluate(dg);
}
```

10.6.3 NOISE IDIOMS AND SPECTRAL SYNTHESIS

The fact that noise is a band-limited function means that its frequency content can be adjusted by scaling the domain over which it is evaluated. For example, if `Noise(p)` has some known frequency content, then the frequency content of `Noise(2*p)` will be twice as high. This is just like the relationship between the frequency content of $\sin(x)$ and $\sin(2x)$. This technique can be used to create a noise function with a desired rate of variation.

For many applications in procedural texturing, it's useful to have variation over multiple scales—for example, to add finer variations to the base noise function. One effective way to do this with noise is to compute patterns via *spectral synthesis*, where a complex function $f_s(s)$ is defined by a sum of contributions from another function $f(x)$:

$$f_s(x) = \sum_i w_i f(s_i x),$$

for a set of weight values w_i and parameter scale values s_i. If the base function $f(x)$ has a well-defined frequency content (e.g., is a sine or cosine function, or a noise function), then each term $f(s_i x)$ also has a well-defined frequency content as described earlier. Because each term of the sum is weighted by a weight value w_i, the result is a sum of contributions of various frequencies, with different frequency ranges weighted differently.

Typically, the scales s_i are chosen in a geometric progression such that $s_i = 2s_{i-1}$ and the weights are $w_i = w_{i-1}/2$. The result is that as higher-frequency variation is added to

DotsTexture::insideDot 559
Noise() 555
Texture::Evaluate() 520

the function, it has relatively less influence on the overall shape of $f_s(x)$. Each additional term is called an *octave* of noise, since it has twice the frequency content of the previous one. When this scheme is used with Perlin noise, the result is often referred to as "fractional Brownian motion" (fBm), after a particular type of random process that varies in a similar manner.

Fractional Brownian motion is a useful building block for procedural textures because it gives a function with more complex variation than plain noise, while still being easy to compute and still having well-defined frequency content. The utility function `FBm()` implements the fractional Brownian motion function. Figure 10.28 shows two graphs of it.

In addition to the point at which to evaluate the function and the function's partial derivatives at that point, the function takes an `omega` parameter, which ranges from zero to one and affects the smoothness of the pattern by controlling the falloff of contributions at higher frequencies (values around 0.5 work well), and `maxOctaves`, which gives the maximum number of octaves of noise that should be used in computing the sum.

⟨*Texture Method Definitions*⟩ +≡

```
float FBm(const Point &P, const Vector &dpdx, const Vector &dpdy,
          float omega, int maxOctaves) {
    ⟨Compute number of octaves for antialiased FBm 562⟩
    ⟨Compute sum of octaves of noise for FBm 563⟩
    return sum;
}
```

Antialiasing the fBm function is based on a technique called *clamping* (Norton, Rockwood, and Skolmoski 1982). The idea is that when we are computing a value based on a sum of components, each with known frequency content, we should stop adding in components that would have frequencies beyond the Nyquist limit and instead add their average values to the sum. Because the average value of `Noise()` is zero, all that needs to be done is to compute the number of octaves such that none of the terms has excessively high frequencies and not evaluate the noise function for any higher octaves.

`Noise()` (and thus the first term of $f_s(x)$ as well) has a maximum frequency content of roughly $\omega = 1$. Each subsequent term represents a doubling of frequency content. Therefore, we would like to find the appropriate number of terms n such that if the sampling rate in noise space is s, we have

$$2^n s = 2\omega = 2.$$

This condition guarantees that there are no frequencies that can't be represented at the sampling rate. Thus, we have

$$2^{n-1} = \frac{1}{s}$$

$$n - 1 = \log\left(\frac{1}{s}\right)$$

$$n = 1 - \frac{1}{2}\log(s^2).$$

FBm() 561
Noise() 555
Point 63
Vector 57

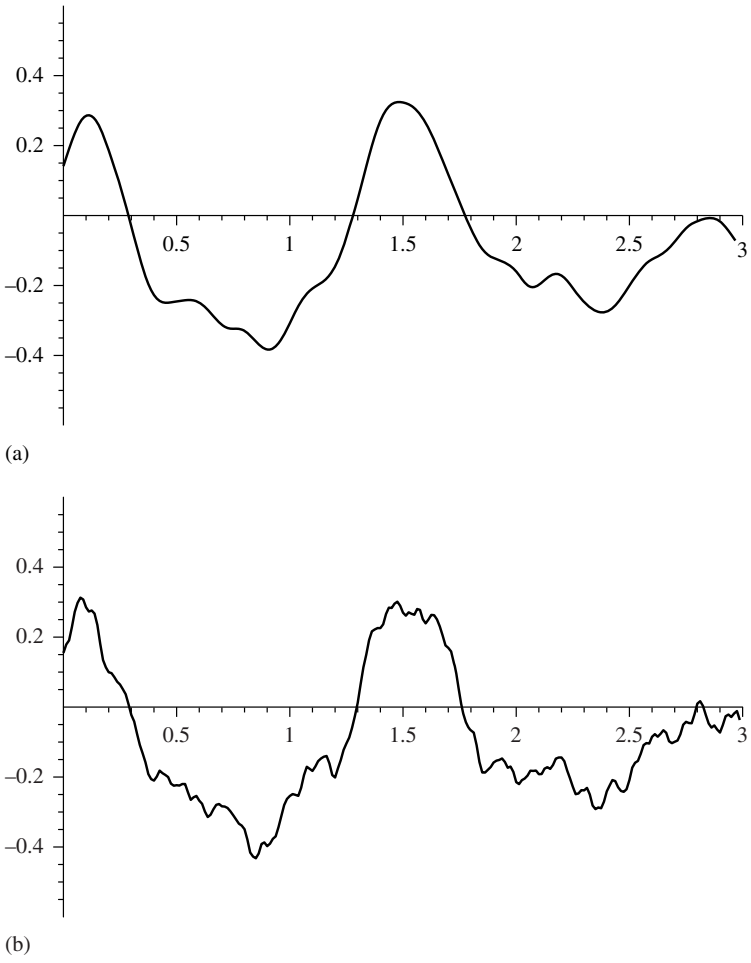


Figure 10.28: Graphs of the `FBm()` Function. (a) 2 and (b) 6 octaves of noise. Notice how as more levels of noise are added, the graph has progressively more detail, although its overall shape remains roughly the same.

The squared sampling rate s^2 can be computed by finding the maximum of the length of the differentials $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$.

⟨*Compute number of octaves for antialiased FBm*⟩ ≡ **561, 564**

```
float s2 = max(dpdx.LengthSquared(), dpdy.LengthSquared());
float foctaves = min((float)maxOctaves, 1.f - .5f * Log2(s2));
int octaves = Floor2Int(foctaves);
```

`FBm()` 561
`Floor2Int()` 1002
`Log2()` 1001
`Vector::LengthSquared()` 62

Finally, the integral number of octaves up to the Nyquist limit are added together and the last octave is faded in, according to the fractional part of `foctaves`. This ensures that

successive octaves of noise fade in gradually, rather than appearing abruptly, which can cause visually noticeable artifacts at the transitions. The implementation here actually increases the frequency between octaves by 1.99, rather than by a factor of 2, in order to reduce the impact of the fact that the noise function is zero at integer lattice points. This breaks up that regularity across sums of octaves of noise, which can also lead to subtle visual artifacts.

⟨*Compute sum of octaves of noise for FBm*⟩ ≡ 561
```
float sum = 0., lambda = 1., o = 1.;
for (int i = 0; i < octaves; ++i) {
    sum += o * Noise(lambda * P);
    lambda *= 1.99f;
    o *= omega;
}
float partialOctave = foctaves - octaves;
sum += o * SmoothStep(.3f, .7f, partialOctave) * Noise(lambda * P);
```

The `SmoothStep()` function takes a minimum and maximum value and a point at which to evaluate a smooth interpolating function. If the point is below the minimum zero is returned, and if it's above the maximum one is returned. Otherwise, it smoothly interpolates between zero and one using a cubic Hermite spline.

⟨*Texture Inline Functions*⟩ ≡
```
inline float SmoothStep(float min, float max, float value) {
    float v = Clamp((value - min) / (max - min), 0.f, 1.f);
    return v * v * (-2.f * v  + 3.f);
}
```

Closely related to the `FBm()` function is the `Turbulence()` function. It also computes a sum of terms of the noise function, but takes the absolute value of each one:

$$f_s(x) = \sum_i w_i |f(s_i x)|.$$

Taking the absolute value introduces first-derivative discontinuities in the synthesized function and thus the turbulence function has infinite frequency content. Nevertheless, the visual characteristics of this function can be quite useful for procedural textures. Figure 10.29 shows two graphs of the turbulence function.

The `Turbulence()` implementation here tries to antialias itself in the same way that `FBm()` did. As described earlier, however, the first-derivative discontinuities in the function introduce infinitely high-frequency content, so these efforts can't hope to be perfectly successful. The `Turbulence()` antialiasing here at least eliminates some of the worst of the artifacts; otherwise, increasing the pixel sampling rate is the best recourse. In practice, this function doesn't alias too terribly when used in procedural textures, particularly compared to the aliasing from infinitely high frequencies from geometric and shadow edges.

Clamp() 1000
FBm() 561
Noise() 555
SmoothStep() 563
Turbulence() 564

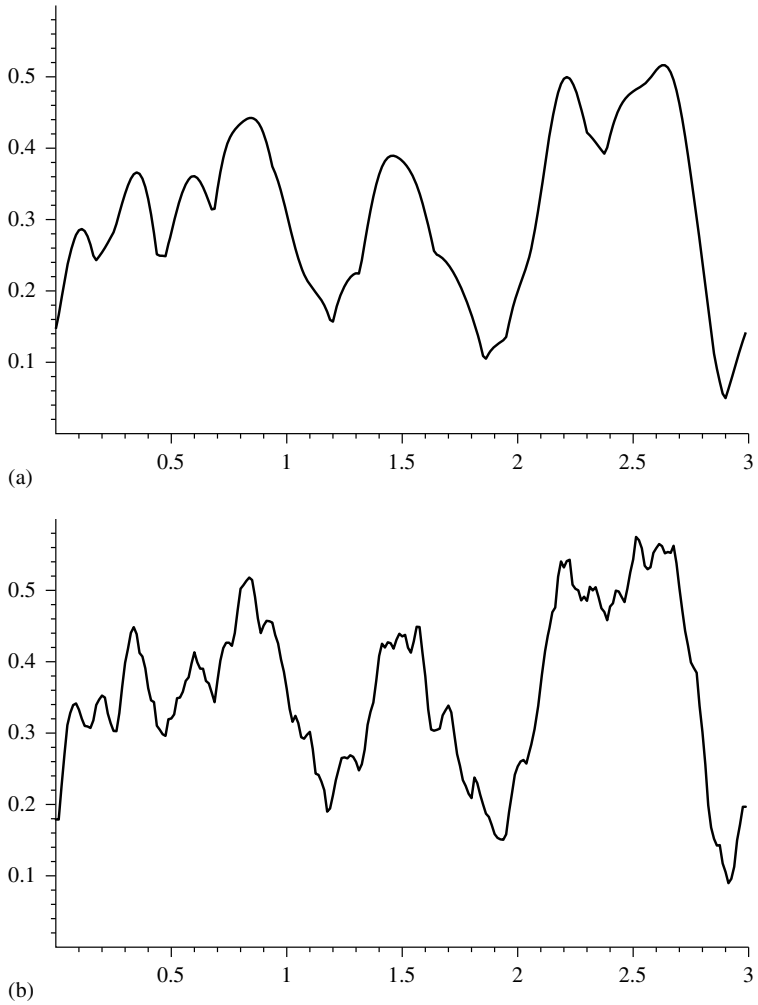


Figure 10.29: Graphs of the `Turbulence()` function for (a) 2 and (b) 6 octaves of noise. Note that the first derivative discontinuities introduced by taking the absolute value of the noise function make this function substantially more rough than FBm.

⟨*Texture Method Definitions*⟩ +≡

```
float Turbulence(const Point &P, const Vector &dpdx, const Vector &dpdy,
                 float omega, int maxOctaves) {
    ⟨Compute number of octaves for antialiased FBm 562⟩
    ⟨Compute sum of octaves of noise for turbulence 565⟩
    return sum;
}
```

Point 63
Turbulence() 564
Vector 57

⟨*Compute sum of octaves of noise for turbulence*⟩ ≡ 564
```
float sum = 0., lambda = 1., o = 1.;
for (int i = 0; i < octaves; ++i) {
    sum += o * fabsf(Noise(lambda * P));
    lambda *= 1.99f;
    o *= omega;
}
float partialOctave = foctaves - octaves;
sum += o * SmoothStep(.3f, .7f, partialOctave) *
       fabsf(Noise(lambda * P));
```

10.6.4 BUMPY AND WRINKLED TEXTURES

The fBm and turbulence functions are particularly useful as a source of random variation for bump mapping. The `FBmTexture` is a `float`-valued texture that uses `FBm()` to compute offsets, and `WrinkledTexture` uses `Turbulence()` to do so. They are demonstrated in Figures 10.30 and 10.31 and are implemented in `textures/fbm.h`, `textures/fbm.cpp`, `textures/wrinkled.h`, and `textures/wrinkled.cpp`.

⟨*FBmTexture Declarations*⟩ ≡
```
template <typename T> class FBmTexture : public Texture<T> {
public:
    ⟨FBmTexture Public Methods 566⟩
private:
    ⟨FBmTexture Private Data 566⟩
};
```

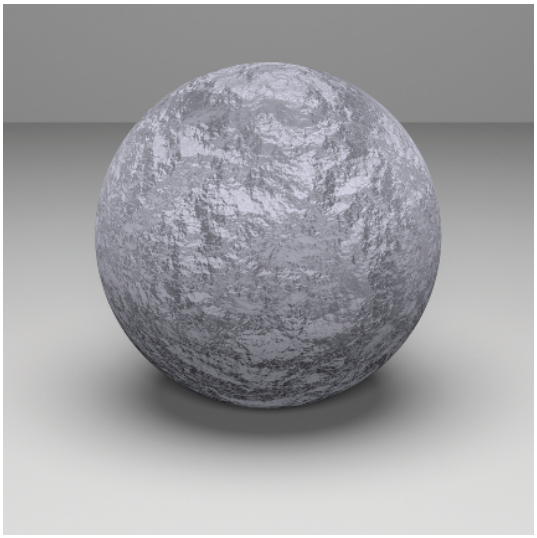


Figure 10.30: Sphere with `FBmTexture` Used for Bump Mapping.

FBm() 561
FBmTexture 565
Noise() 555
SmoothStep() 563
Texture 519
Turbulence() 564
WrinkledTexture 566

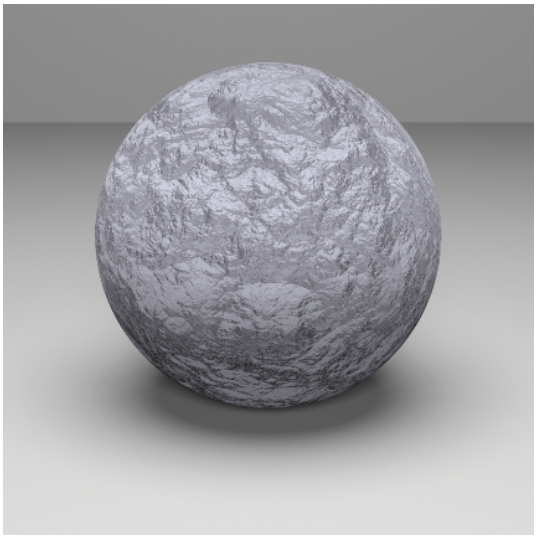


Figure 10.31: `WrinkledTexture` Used as Bump Mapping Function for Sphere.

⟨*FBmTexture Public Methods*⟩ ≡ 565
```
FBmTexture(int oct, float roughness, TextureMapping3D *map)
    : omega(roughness), octaves(oct), mapping(map) { }
```

⟨*FBmTexture Private Data*⟩ ≡ 565
```
float omega;
int octaves;
TextureMapping3D *mapping;
```

⟨*FBmTexture Public Methods*⟩ +≡ 565
```
T Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    return FBm(P, dpdx, dpdy, omega, octaves);
}
```

The implementation of `WrinkledTexture` is almost identical to `FBmTexture`, save for a call to `Turbulence()` instead of `FBm()`. As such, it isn't included here.

10.6.5 WINDY WAVES

Application of fBm can give a reasonably convincing representation of waves (Ebert et al. 2003). Figures 1.11, 4.1, and 7.36 use this texture for the water in those scenes. This `Texture` is based on two observations. First, across the surface of a wind-swept lake (for example), some areas are relatively smooth and some are more choppy; this effect comes from the natural variation of the wind's strength from area to area. Second, the overall

DifferentialGeometry 102
FBm() 561
FBmTexture 565
FBmTexture::mapping 566
FBmTexture::octaves 566
FBmTexture::omega 566
Point 63
Texture 519
TextureMapping3D 519
TextureMapping3D::Map() 519
Turbulence() 564
Vector 57
WrinkledTexture 566

form of individual waves on the surface can be described well by the fBm-based wave pattern scaled by the wind strength. This texture is implemented in `textures/windy.h` and `textures/windy.cpp`.

⟨*WindyTexture Declarations*⟩ ≡
```
template <typename T> class WindyTexture : public Texture<T> {
public:
    ⟨WindyTexture Public Methods 567⟩
private:
    ⟨WindyTexture Private Data 567⟩
};
```

⟨*WindyTexture Public Methods*⟩ ≡ 567
```
WindyTexture(TextureMapping3D *map) : mapping(map) { }
```

⟨*WindyTexture Private Data*⟩ ≡ 567
```
TextureMapping3D *mapping;
```

The evaluation function uses two calls to the `FBm()` function. The first scales down the point `P` by a factor of 10; as a result, the first call to `FBm()` returns relatively low-frequency variation over the surface of the object being shaded. This value is used to determine the local strength of the wind. The second call determines the amplitude of the wave at the particular point, independent of the amount of wind there. The product of these two values gives the actual wave offset for the particular location.

⟨*WindyTexture Public Methods*⟩ +≡ 567
```
T Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    float windStrength = FBm(.1f * P, .1f * dpdx, .1f * dpdy, .5f, 3);
    float waveHeight = FBm(P, dpdx, dpdy, .5f, 6);
    return fabsf(windStrength) * waveHeight;
}
```

DifferentialGeometry 102
FBm() 561
MarbleTexture 568
Point 63
Texture 519
TextureMapping3D 519
TextureMapping3D::Map() 519
Vector 57
WindyTexture 567
WindyTexture::mapping 567

10.6.6 MARBLE

Another classic use of the noise function is to perturb texture coordinates before using another texture or lookup table. For example, a facsimile of marble can be made by modeling the marble material as a series of layered strata and then using noise to perturb the coordinate used for finding a value among the strata. The `MarbleTexture` in this section implements this approach. Figure 10.32 illustrates the idea behind this texture. On the left, the layers of marble are indexed directly using the y coordinate of the point on the sphere. On the right, fBm has been used to perturb the y value, introducing variation. This texture is implemented in `textures/marble.h` and `textures/marble.cpp`.

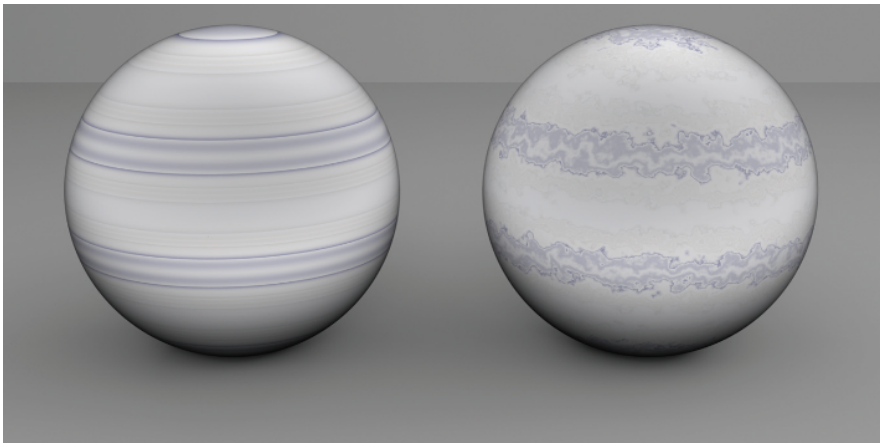


Figure 10.32: Marble. The `MarbleTexture` perturbs the coordinate used to index into a one-dimensional table of colors using FBm, giving a plausible marble appearance.

⟨*MarbleTexture Declarations*⟩ ≡
```
class MarbleTexture : public Texture<Spectrum> {
public:
    ⟨MarbleTexture Public Methods 568⟩
private:
    ⟨MarbleTexture Private Data 568⟩
};
```

The texture takes the usual set of parameters to control the `FBm()` function that will be used to perturb the lookup coordinate. The `variation` parameter modulates the magnitude of the perturbation.

⟨*MarbleTexture Public Methods*⟩ ≡ 568
```
MarbleTexture(int oct, float roughness, float sc, float var,
              TextureMapping3D *map)
    : octaves(oct), omega(roughness), scale(sc), variation(var),
      mapping(map) { }
```

⟨*MarbleTexture Private Data*⟩ ≡ 568
```
int octaves;
float omega, scale, variation;
TextureMapping3D *mapping;
```

An offset into the marble layers is computed by adding the variation to the point's y component and using the sine function to remap its value into the range [0, 1]. The ⟨*Evaluate marble spline at* `t`⟩ fragment uses the t value as the evaluation point for a cubic spline through a series of colors that are similar to those of real marble.

FBm() 561
MarbleTexture 568
MarbleTexture::mapping 568
MarbleTexture::octaves 568
MarbleTexture::omega 568
MarbleTexture::scale 568
MarbleTexture::variation 568
Spectrum 263
Texture 519
TextureMapping3D 519

⟨*MarbleTexture Public Methods*⟩ +≡ 568

```
Spectrum Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    P *= scale;
    float marble = P.y + variation *
                   FBm(P, scale * dpdx, scale * dpdy, omega, octaves);
    float t = .5f + .5f * sinf(marble);
    ⟨Evaluate marble spline at t⟩
}
```

FURTHER READING

The cone-tracing method of Amanatides (1984) was one of the first techniques for automatically estimating filter footprints for ray tracing. The beam-tracing algorithm of Heckbert and Hanrahan (1984) was another early extension of ray tracing to incorporate an area associated with each image sample, rather than just an infinitesimal ray. The pencil-tracing method of Shinya, Takahashi, and Naito (1987) is another approach to this problem. Other related work on the topic of associating areas or footprints with rays includes Mitchell and Hanrahan's paper on rendering caustics (Mitchell and Hanrahan 1992) and Turkowski's technical report (Turkowski 1993).

Collins (1994) estimated the ray footprint by keeping a tree of all rays traced from a given camera ray, examining corresponding rays at the same level and position. Also, Worley's chapter in *Texturing and Modeling* (Ebert et al. 2003) on computing differentials for filter regions presents an approach similar to ours. The ray differentials used in `pbrt` are based on Igehy's (1999) formulation, which was extended by Suykens and Willems (2001) to handle glossy reflection in addition to perfect specular reflection.

Two-dimensional texture mapping with images was first introduced to graphics by Blinn and Newell (1976). Ever since Crow (1977) identified aliasing as the source of many errors in images in graphics, quite a bit of work has been done to find efficient and effective ways of antialiasing image maps. Dungan, Stenger, and Sutty (1978) were the first to suggest creating a pyramid of prefiltered texture images; they used the nearest texture sample at the appropriate level when looking up texture values, using supersampling in screen space to antialias the result. Feibush, Levoy, and Cook (1980) investigated a spatially varying filter function, rather than a simple box filter. (Blinn and Newell were aware of Crow's results and used a box filter for their textures.)

Williams (1983) used a MIP map image pyramid for texture filtering with trilinear interpolation. Shortly thereafter, Crow (1984) introduced summed area tables, which make it possible to efficiently filter over axis-aligned rectangular regions of texture space. Summed area tables handle anisotropy better than Williams's method, although only for primarily axis-aligned filter regions. Heckbert (1986) wrote a good general survey of texture mapping algorithms through the mid-1980s.

Greene and Heckbert (1986) originally developed the elliptically weighted average technique, and Heckbert's master's thesis put the method on a solid theoretical footing

DifferentialGeometry 102
FBm() 561
MarbleTexture::mapping 568
Point 63
Spectrum 263
TextureMapping3D::Map() 519
Vector 57

(Heckbert 1989). Fournier and Fiume (1988) developed an even higher-quality texture filtering method that focuses on using a bounded amount of computation per lookup. Nonetheless, their method appears to be less efficient than EWA overall. Lansdale's master's thesis has an extensive description of EWA and Fournier and Fiume's method, including implementation details (Lansdale 1991).

More recently, a number of researchers have investigated generalizing Williams's original method using a series of trilinear MIP map samples in an effort to increase quality without having to pay the price for the general EWA algorithm. By taking multiple samples from the MIP map, anisotropy is handled well while preserving the computational efficiency. Examples include Barkans's (1997) description of texture filtering in the Talisman architecture, McCormack et al.'s (1999) Feline method, and Cant and Shrubsole's (2000) technique. Chen et al. (2004) summarized the state of the art in texture filtering and introduced a method that addresses some of the shortcomings of existing techniques.

Gamma correction has a long history in computer graphics. Poynton (2002a, 2002b) has written comprehensive FAQs on issues related to color representation and gamma correction. Even though there is no physical reason for LCD displays to have a nonlinear response, Fairchild and Wyble (1998) measured the response of a widely used LCD and found that gamma correction is also necessary with that display. See also Gibson and Fairchild (2000), where similar results are found for other LCDs. See Gritz and d'Eon (2008) for a detailed discussion of the implications of gamma correction for rendering and how to correctly account for it in rendering systems.

Smith's Web site and document on audio resampling gives a good overview of resampling signals in one dimension (Smith 2002). Heckbert's `zoom` source code is the canonical reference for image resampling (Heckbert 1989). His implementation carefully avoids feedback without using auxiliary storage, unlike ours in this chapter, which allocates additional temporary buffer space to do so.

Three-dimensional solid texturing was originally developed by Gardner (1984, 1985), Perlin (1985a), and Peachey (1985). Norton, Rockwood, and Skolmoski (1982) developed the *clamping* method that is widely used for antialiasing textures based on solid texturing. The general idea of procedural texturing was introduced by Cook (1984), Perlin (1985a), and Peachey (1985).

Peachey's chapter in *Texturing and Modeling* (Ebert et al. 2003) has a thorough summary of approaches to noise functions. After Perlin's original noise function, both Lewis (1989) and van Wijk (1991) developed alternatives that made different time/quality trade-offs. Worley (1996) has developed a quite different noise function for procedural texturing that is well suited for cellular and organic patterns. Perlin (2002) revised his noise function to correct a number of subtle shortcomings.

Noise functions have received additional attention from the research community in recent years. Building on Lewis's observation that individual bands of Perlin's noise function actually have frequency content over a fairly wide range (Lewis 1989), Cook and DeRose (2005) also identified the problem that 2D slices through 3D noise functions aren't in general band limited, even if the original 3D noise function is. They proposed a new noise function that addresses both of these issues. Goldberg et al. (2008) developed a noise function that makes efficient anisotropic filtering possible, leading to higher-

quality results than just applying the clamping approach for antialiasing. Their method is also well suited to programmable graphics hardware. Kensler et al. (2008) suggested a number of improvements to Perlin's revised noise function. Finally, Lagae et al. (2009) have developed a noise function that has good frequency control and can be mapped well to surfaces even without a surface parameterization.

The first languages and systems that supported the idea of user-supplied procedural shaders were developed by Cook (1984) and Perlin (1985a). (The texture composition model in this chapter is similar to Cook's shade trees.) The RenderMan shading language, described in a paper by Hanrahan and Lawson (1990), remains the classic shading language in graphics. See Ebert et al. (2003) and Apodaca and Gritz (2000) for techniques for writing procedural shaders; both of those have excellent discussions of issues related to antialiasing in procedural shaders. More recently, Van Horn and Turk (2008) developed an approach to automatically generate MIP maps of reflection functions that represent the characteristics of shaders over finite areas in order to antialias them.

Many creative methods for computing texture on surfaces have been developed. A sampling of our favorites includes reaction diffusion, which simulates growth processes based on a model of chemical interactions over surfaces and was simultaneously introduced by Turk (1991) and Witkin and Kass (1991); Sims's genetic algorithm-based approach, which finds programs that generate interesting textures through random mutations from which a user selects their favorites (Sims 1991); Fleischer et al.'s cellular texturing algorithms that generate geometrically accurate scales and spike features on surfaces (Fleischer et al. 1995); and Dorsey et al.'s flow simulations that model the effect of weathering on buildings and encode the results in image maps that stored the relative wetness, dirtiness, and so on, at points on the surfaces of structures (Dorsey et al. 1996). Porumbescu et al. (2005) developed *shell maps*, which make it possible to map geometric objects onto a surface in the manner of texture mapping.

A variety of *texture synthesis* algorithms have been developed in the last decade; these approaches take an example texture image and then synthesize larger texture maps that appear similar to the original texture while not being exactly the same. The survey article by Wei et al. (2009) describes recent work in this area as well as the main approaches that have been developed so far.

EXERCISES

❷ 10.1 Many image file formats don't store floating-point color values but instead use eight bits for each color component, mapping the values to the range [0, 1]. For images originally stored in this format, the `ImageTexture` uses four times more memory than strictly necessary by using `floats` in `RGBSpectrum` objects to store these colors. Modify the image reading routines to support an image file format with eight-bit components and return an indication of when an image is read from such a file. Then, modify the `ImageTexture` so that it keeps the data for such textures in an eight-bit representation and modify the `MIPMap` so that it can filter data stored in this format. How much memory is saved for image texture-heavy scenes? How is `pbrt`'s performance affected? Can you explain the causes of any performance differences?

`ImageTexture` 524
`MIPMap` 530
`RGBSpectrum` 279

10.2 For scenes with many image textures where reading them all into memory simultaneously has a prohibitive memory cost, an effective approach can be to allocate a fixed amount of memory for image maps (a *texture cache*), load textures into that memory on demand, and discard the image maps that haven't been accessed recently when the memory fills up (Peachey 1990). To enable good performance with small texture caches, image maps should be stored in a *tiled* format that makes it possible to load in small square regions of the texture independently of each other. Tiling techniques like these are used in graphics hardware to improve the performance of their texture memory caches (Hakura and Gupta 1997; Igehy et al. 1998; Igehy et al. 1999). Implement a texture cache in `pbrt`. Write a conversion program that converts images in other formats to a tiled format. (You may want to investigate OpenEXR's tiled image support.) How small can you make the texture cache and still see good performance?

10.3 The Feline texture filtering technique is a middle ground between trilinear interpolation and EWA filtering. It gives results nearly as good as EWA by doing trilinear filtering at a series of positions along the longer filtering axis in texture space. Read the paper that describes Feline (McCormack et al. 1999) and implement this method in `pbrt`. How do its performance and quality compare to EWA?

10.4 Improve the filtering algorithm used for resampling image maps to initialize the MIP map levels using the Lanczos filter instead of the box filter. How do the spheres test images in the file `scenes/sphere-ewa-vs-trilerp.pbrt` and Figure 10.10 change after your improvements?

10.5 It is possible to use MIP-mapping with textures that have non-power-of-two resolutions—the details are explained by Guthe and Heckbert (2005). Implementing this approach can save a substantial amount of memory: in the worst case, the resampling that `pbrt`'s `MIPMap` implementation performs can increase memory requirements by a factor of four. (Consider a 513×513 texture that is resampled to be 1024×1024.) Implement this approach in `pbrt` and compare the amount of memory used to store texture data for a variety of texture-heavy scenes.

10.6 Some of the light transport algorithms in Chapter 15 require a large number of samples to be taken per pixel for good results. (Examples of such algorithms include path tracing as implemented by the `PathIntegrator`.) If hundreds or thousands of samples are taken in each pixel, then the computational expense of high-quality texture filtering isn't worthwhile; the high pixel sampling rate serves well to antialias texture functions with high frequencies. Modify the `MIPMap` implementation so that it optionally just returns a bilinearly interpolated value from the finest level of the pyramid, even if a filter footprint is provided. Compare rendering time and image quality with this approach when rendering an image using many samples per pixel and a scene that has image maps that would otherwise exhibit aliasing at lower pixel sampling rates.

`MIPMap` 530

`PathIntegrator` 766

10.7 An additional advantage of properly antialiased image map lookups is that they improve cache performance. Consider, for example, the situation of undersampling a high-resolution image map: nearby samples on the screen will access

widely separated parts of the image map, such that there is low probability that texels fetched from main memory for one texture lookup will already be in the cache for texture lookups at adjacent pixel samples. Modify `pbrt` so that it always does image texture lookups from the finest level of the `MIPMap`, being careful to ensure that the same number of texels are still being accessed. How does performance change? What do cache-profiling tools report about the overall change in effectiveness of the CPU cache?

● 10.8 Read Worley's paper that describes a new noise function with substantially different visual characteristics than Perlin noise (Worley 1996). Implement this cellular noise function and add `Textures` to `pbrt` that are based on it.

● 10.9 Implement one of the improved noise functions, such as the ones introduced by Cook and DeRose (2005), Goldberg et al. (2008), or Lagae et al. (2009). Compare image quality and rendering time for scenes that make substantial use of noise functions to the current implementation in `pbrt`.

● 10.10 The implementation of the `DotsTexture` texture in this chapter does not make any effort to avoid aliasing in the results that it computes. Modify this texture to do some form of antialiasing. The `Checkerboard2DTexture` offers a guide as to how this might be done, although this case is more complicated, both because the polka dots are not present in every grid cell and because they are irregularly positioned.

At the two extremes of a filter region that is within a single cell and a filter region that spans a large number of cells, the task is easier. If the filter is entirely within a single cell and is entirely inside or outside the polka dot in that cell (if present), then it is only necessary to evaluate one of the two subtextures as appropriate. If the filter is within a single cell but overlaps both the dot and the base texture, then it is possible to compute how much of the filter area is inside the dot and how much is outside and blend between the two. At the other extreme, if the filter area is extremely large, it is possible to blend between the two textures according to the overall average of how much area is covered by dots and how much is not. (Note that this approach potentially makes the same error as was made in the checkerboard, where the subtextures aren't aware that part of their area is occluded by another texture. Ignore this issue for this exercise.)

Implement these approaches and then consider the intermediate cases, where the filter region spans a small number of cells. What approaches work well for antialiasing in this case?

● 10.11 Write a general-purpose `Texture` that stores a reference to another texture and supersamples that texture when the evaluation method is called, thus making it possible to apply supersampling to any `Texture`. Use your implementation to compare the effectiveness and quality of the built-in antialiasing done by various procedural textures. Also compare the run time efficiency of texture supersampling versus increased pixel sampling.

Checkerboard2DTexture 547
DotsTexture 559
Texture 519

■ 10.12 Modify `pbrt` to support a shading language to allow user-written programs to compute texture values.



CHAPTER ELEVEN

★ 11 VOLUME SCATTERING

So far, we have assumed that scenes are made up of collections of surfaces in a vacuum, which means that radiance is constant along rays between surfaces. However, there are many real-world situations where this assumption is inaccurate: fog and smoke attenuate and scatter light, and scattering from particles in the atmosphere makes the sky blue and sunsets red. This chapter introduces the mathematics to describe how light is affected as it passes through *participating media*—particles distributed throughout a region of 3D space. Simulating the effect of participating media makes it possible to render images with atmospheric haze, beams of light through clouds, light passing through cloudy water, and subsurface scattering, where light exits a solid object at a different place than where it entered.

This chapter first describes the basic physical processes that affect the radiance along rays passing through participating media. It then introduces the `VolumeRegion` base class, an interface for modeling different types of media. Like a `BSDF`, the volume description characterizes how light is scattered at individual points. In order to determine the global effect on the distribution of light in the scene, `VolumeIntegrators` are necessary; they are described in Chapter 16. The chapter concludes with the abstraction that represents the subsurface scattering properties of objects, the `BSSRDF`, as well a number of `Materials` for translucent objects.

11.1 VOLUME SCATTERING PROCESSES

There are three main processes that affect the distribution of radiance in an environment with participating media:

- *Absorption*—the reduction in radiance due to the conversion of light to another form of energy, such as heat
- *Emission*—energy that is added to the environment from luminous particles

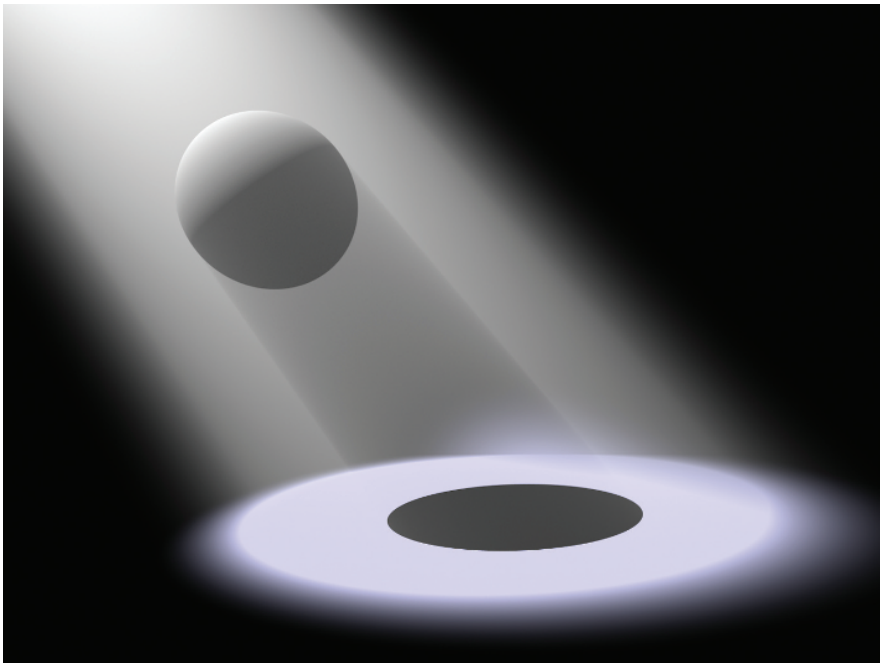


Figure 11.1: Spotlight through Fog. Light scattering from particles in the medium back toward the camera makes the spotlight's illumination visible even in pixels where there are no visible surfaces that reflect it. The sphere blocks light, casting a volumetric shadow in the region beneath it.

- *Scattering*—how light heading in one direction is scattered to other directions due to collisions with particles

The characteristics of all of these properties may be *homogeneous* or *inhomogeneous*. Homogeneous properties are constant throughout a given spatial extent, while inhomogeneous properties may vary throughout space. Figure 11.1 shows a simple example of volume scattering, where a spotlight shining through a participating medium illuminates the medium and and casts a volumetric shadow.

11.1.1 ABSORPTION

Consider thick black smoke from a fire: the smoke obscures the objects behind it because its particles absorb light traveling from the object to the viewer. The thicker the smoke, the more light is absorbed. Figure 11.2 shows this effect with a volume density that was created with an accurate physical simulation of smoke formation. Note the shadow on the ground: the participating medium has also absorbed light between the light source to the ground plane, casting a shadow.

Absorption is described by the medium's *absorption cross section*, σ_a, which is the probability density that light is absorbed per unit distance traveled in the medium. In general, the absorption cross section may vary with both position p and direction ω, although it is normally just a function of position. It is usually also a spectrally varying quantity. The

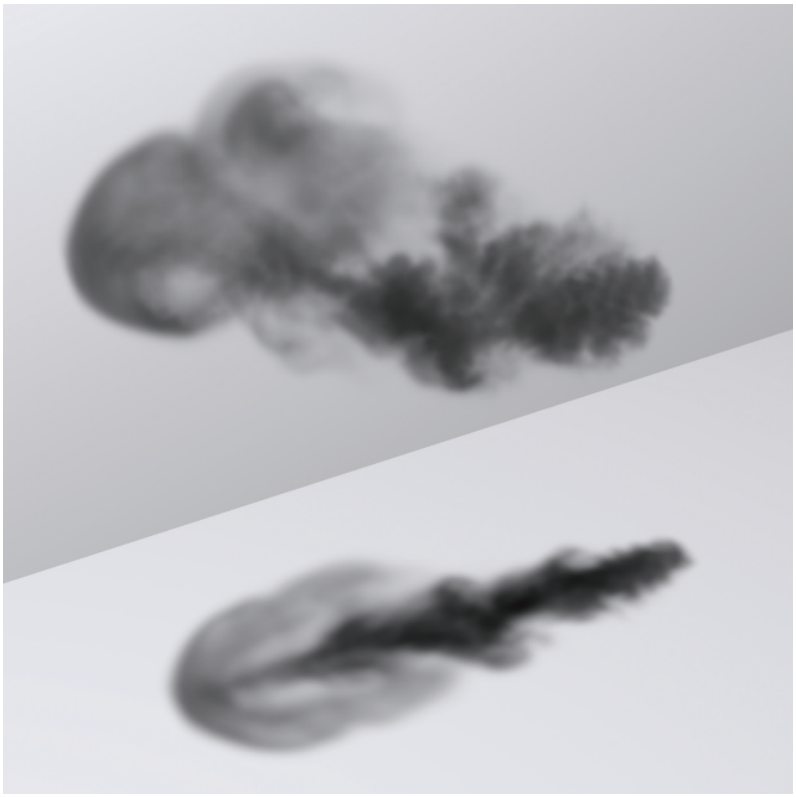


Figure 11.2: If a participating medium primarily absorbs light passing through it, it will have a dark and smoky appearance, as shown here. *(Smoke simulation data courtesy of Duc Nguyen and Ron Fedkiw.)*

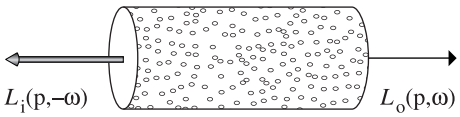


Figure 11.3: Absorption reduces the amount of radiance along a ray through a participating medium. Consider a ray carrying incident radiance at a point p from direction $-\omega$. If the ray passes through a differential cylinder filled with absorbing particles, the change in radiance due to absorption by those particles is $dL_o(p, \omega) = -\sigma_a(p, \omega)L_i(p, -\omega)dt$.

units of σ_a are reciprocal distance (m^{-1}). This means that σ_a can take on any positive value; it is not required to be between zero and one, for instance.

Figure 11.3 shows the effect of absorption along a very short portion of a ray. Some amount of radiance $L_i(p, -\omega)$ is arriving at point p, and we'd like to find the exitant

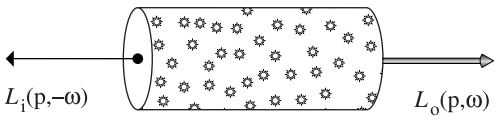


Figure 11.4: The volume emission function $L_{ve}(p, \omega)$ gives the change in radiance along a ray as it passes through a differential volume of emissive particles. The change in radiance per differential distance is $dL = L_{ve}dt$.

radiance $L_o(p, \omega)$ after absorption in the differential volume. This change in radiance along the differential ray length dt is described by the differential equation

$$L_o(p, \omega) - L_i(p, -\omega) = dL_o(p, \omega) = -\sigma_a(p, \omega)\, L_i(p, -\omega)\, dt,$$

which says that the differential reduction in radiance along the beam is a linear function of its initial radiance.[1]

This differential equation can be solved to give the integral equation describing the total fraction of light absorbed for a ray. If we assume that the ray travels a distance d in direction ω through the medium starting at point p, the total absorption is given by

$$e^{-\int_0^d \sigma_a(p+t\omega,\omega)\, dt}.$$

11.1.2 EMISSION

While absorption reduces the amount of radiance along a ray as it passes through a medium, emission increases it, due to chemical, thermal, or nuclear processes that convert energy into visible light. Figure 11.4 shows emission in a differential volume, where we denote emitted radiance added to a ray per unit distance at a point p in direction ω by $L_{ve}(p, \omega)$. Figure 11.5 shows the effect of emission in the smoke data set. In that figure the absorption coefficient is much lower than in Figure 11.2, giving a very different appearance.

The differential equation that gives the change in radiance due to emission is

$$dL_o(p, \omega) = L_{ve}(p, \omega)\, dt.$$

Note that this equation is based on the assumption that the emitted light L_{ve} is not dependent on the incoming light L_i. This is always true under the linear optics assumptions that `pbrt` is based on.

1 This is another instance of the linearity assumption in radiometry: the fraction of light absorbed doesn't vary based on the ray's radiance, but is always a fixed fraction.

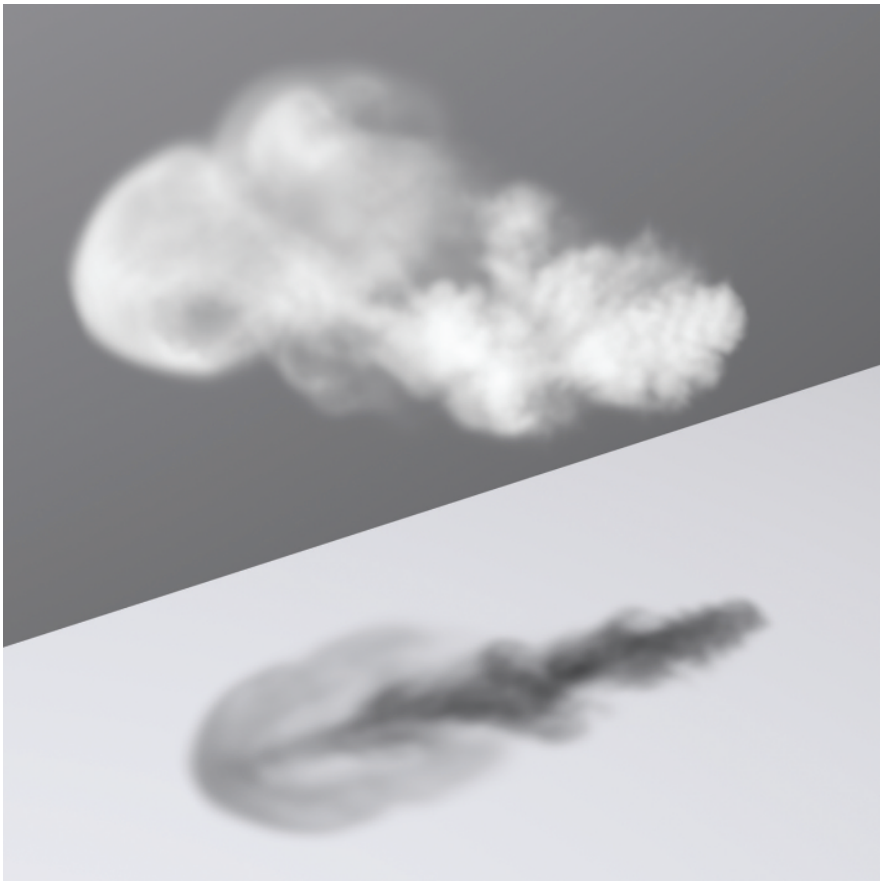


Figure 11.5: A Participating Medium Where the Dominant Volumetric Effect Is Emission. Although the medium still absorbs light, still casting a shadow on the ground and obscuring the wall behind it, emission in the volume increases radiance along rays passing through it, making the cloud brighter than the wall behind it.

11.1.3 OUT-SCATTERING AND EXTINCTION

The third basic light interaction in participating media is scattering. As a beam passes through a medium, it may collide with particles in the medium and be scattered in different directions. This has two effects on the total radiance that the beam carries. It reduces the radiance exiting a differential region of the beam because some of it is deflected to different directions. This effect is called *out-scattering* (Figure 11.6) and is the topic of this section. However, radiance from other rays may be scattered into the path of the current ray; this *in-scattering* process is the subject of the next section.

The probability of an out-scattering event occurring per unit distance is given by the scattering coefficient, σ_s. As with the attenuation coefficient, the reduction in radiance

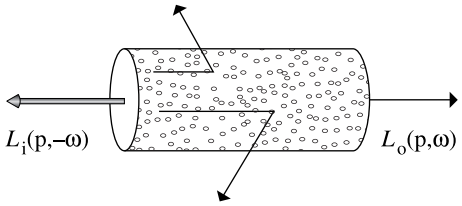


Figure 11.6: Like absorption, out-scattering also reduces the radiance along a ray. Light that hits particles may be scattered in another direction such that the radiance exiting the region in the original direction is reduced.

along a differential length dt due to out-scattering is given by

$$dL_o(p, \omega) = -\sigma_s(p, \omega)\, L_i(p, -\omega)\, dt.$$

The total reduction in radiance due to absorption and out-scattering is given by the sum $\sigma_a + \sigma_s$. This combined effect of absorption and out-scattering is called *attenuation* or *extinction*. For convenience the sum of these two coefficients is denoted by the attenuation coefficient σ_t:

$$\sigma_t(p, \omega) = \sigma_a(p, \omega) + \sigma_s(p, \omega).$$

Given the attenuation coefficient σ_t, the differential equation describing overall attenuation,

$$\frac{dL_o(p, \omega)}{dt} = -\sigma_t(p, \omega)\, L_i(p, -\omega),$$

can be solved to find the *beam transmittance*, which gives the fraction of radiance that is transmitted between two points on a ray:

$$T_r(p \to p') = e^{-\int_0^d \sigma_t(p+t\omega, \omega)\, dt}$$

where d is the distance between p and p′, ω is the normalized direction vector between them, and T_r denotes the beam transmittance between p and p′. Note that the transmittance is always between zero and one. Thus, if exitant radiance from a point p on a surface in a given direction ω is given by $L_o(p, \omega)$, after accounting for extinction, the incident radiance at another point p′ in direction $-\omega$ is

$$T_r(p \to p')L_o(p, \omega).$$

This idea is illustrated in Figure 11.7.

Two useful properties of beam transmittance are that transmittance from a point to itself is one ($T_r(p \to p) = 1$), and in a vacuum $T_r(p \to p') = 1$ for all p′. Another important property, true in all media, is that transmittance is multiplicative along points on a ray:

$$T_r(p \to p'') = T_r(p \to p')T_r(p' \to p''),$$

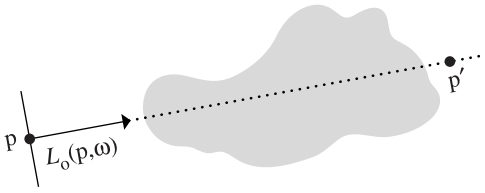


Figure 11.7: The beam transmittance $T_r(\mathrm{p} \to \mathrm{p}')$ gives the fraction of light transmitted from one point to another, accounting for absorption and out-scattering, but ignoring emission and in-scattering. Given exitant radiance at a point p in direction ω (e.g., reflected radiance from a surface), the radiance visible at another point p′ along the ray is $T_r(\mathrm{p} \to \mathrm{p}')L_o(\mathrm{p}, \omega)$.

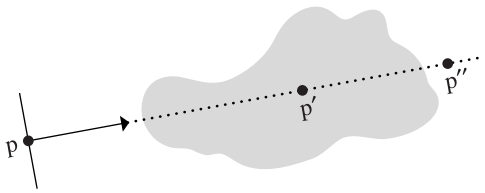


Figure 11.8: A useful property of beam transmittance is that it is multiplicative: the transmittance between points p and p″ on a ray like the one shown here is equal to the transmittance from p to p′ times the transmittance from p′ to p″ for all points p′ between p and p″.

for all points p′ between p and p″ (Figure 11.8). This property is important for volume scattering implementations, since it makes it possible to incrementally compute transmittance at many points along a ray by computing the product of each previously computed transmittance with the transmittance for its next segment.

The negated exponent in T_r is called the *optical thickness* between the two points. It is denoted by the symbol τ:

$$\tau(\mathrm{p} \to \mathrm{p}') = \int_0^d \sigma_t(\mathrm{p} + t\omega, -\omega)\, \mathrm{d}t.$$

In a homogeneous medium, σ_t is a constant, so the τ integral is trivially evaluated and yields *Beer's law:*

$$T_r(\mathrm{p} \to \mathrm{p}') = \mathrm{e}^{-\sigma_t d}. \tag{11.1}$$

11.1.4 IN-SCATTERING

While out-scattering reduces radiance along a ray due to scattering in different directions, *in-scattering* accounts for increased radiance due to scattering from other directions

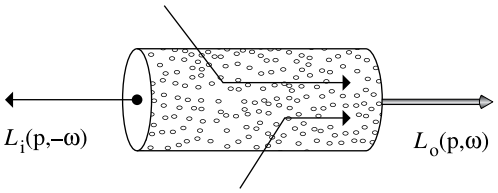


Figure 11.9: In-scattering accounts for the increase in radiance along a ray due to scattering of light from other directions. Radiance from outside the differential volume is scattered along the direction of the ray and added to the incoming radiance.

(Figure 11.9). Figure 11.10 shows the effect of in-scattering with the smoke data set. Note that the smoke appears much thicker than when absorption or emission was the dominant volumetric effect.

Assuming that the separation between particles is at least a few times the lengths of their radii, it is possible to ignore interparticle interactions when describing scattering at a particular location. Under this assumption, the *phase function* $p(\omega \to \omega')$ describes the angular distribution of scattered radiation at a point; it is the volumetric analog to the BSDF. The BSDF analogy is not exact, however; for example, phase functions have a normalization constraint: for all ω, the condition

$$\int_{S^2} p(\omega \to \omega')\, d\omega' = 1 \tag{11.2}$$

must hold. This constraint means that phase functions actually define probability distributions for scattering in a particular direction.

The total added radiance per unit distance due to in-scattering is given by the *source term* S:

$$dL_o(p, \omega) = S(p, \omega)\, dt.$$

It accounts for both volume emission and in-scattering:

$$S(p, \omega) = L_{ve}(p, \omega) + \sigma_s(p, \omega) \int_{S^2} p(p, -\omega' \to \omega)\, L_i(p, \omega')\, d\omega'.$$

The in-scattering portion of the source term is the product of the scattering probability per unit distance, σ_s, and the amount of added radiance at a point, which is given by the spherical integral of the product of incident radiance and the phase function. Note that the source term is very similar to the scattering equation, Equation (5.8); the main difference is that there is no cosine term since the phase function operates on radiance rather than differential irradiance.



Figure 11.10: In-Scattering with the Smoke Data Set. Note the substantially different appearance compared to the other two smoke images.

11.2 PHASE FUNCTIONS

Just as there is a wide variety of BSDF models to describe scattering from surfaces, many phase functions have been developed. These range from parameterized models (which can be used to fit a function with a small number of parameters to measured data) to analytic models that are based on deriving the scattered radiance distribution that results from particles with known shape and material (e.g., spherical water droplets).

In most naturally occurring media, the phase function is a 1D function of only the angle θ between the two directions ω and ω'; such media are called *isotropic*, and these phase functions are often written as $p(\cos\theta)$. In exotic media, such as those with a crystalline structure, the phase function is a 4D function of the two directions. In addition to being normalized, an important property of naturally occurring phase functions is that they are *reciprocal*: the two directions can be interchanged and the phase function's value remains unchanged. Note that isotropic phase functions are trivially reciprocal because $\cos(-\theta) = \cos(\theta)$.

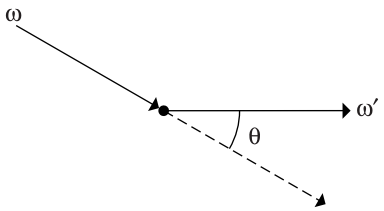


Figure 11.11: Phase functions are written with the convention that the incident direction points toward the point where scattering happens and the outgoing direction ω' points away from it. This is a different convention than was used for BSDFs. The angle between them is denoted by θ.

In a slightly confusing overloading of terminology, phase functions themselves can be isotropic or anisotropic as well. An isotropic phase function describes equal scattering in all directions and is thus independent of either of the two directions. Because phase functions are normalized, there is only one such function, and it must be the constant $1/4\pi$:

$$p_{\text{isotropic}}(\omega \to \omega') = \frac{1}{4\pi}.$$

⟨*Volume Scattering Definitions*⟩ ≡
```
float PhaseIsotropic(const Vector &, const Vector &) {
    return 1.f / (4.f * M_PI);
}
```

All of the anisotropic phase functions in the remainder of this section describe isotropic media and are thus defined in terms of the angle between the two directions (Figure 11.11). Phase functions use a different convention for the direction of the vectors at a scattering event than was used for scattering at a surface, where both vectors faced away from the surface. For phase function implementations in `pbrt`, the incident direction points toward the scattering point and the outgoing direction points away from it. This matches the usual convention for phase functions.

`pbrt` includes implementations of phase functions that model Rayleigh scattering and Mie scattering. The Rayleigh model describes scattering from very small particles such as the molecules in the Earth's atmosphere. If the particles have radii that are smaller than the wavelength of light λ, ($r/\lambda < 0.05$), the Rayleigh model accurately describes the distribution of scattered light. Wavelength-dependent Rayleigh scattering is the reason that the sky is blue and sunsets are red. Mie scattering is based on a more general theory; it is derived from Maxwell's equations and can describe scattering from a wider range of particle sizes. For example, it is a good model for scattering in the atmosphere due to water droplets and fog.

M_PI 1002
Vector 57

We won't include the implementations of these phase functions in the text here. They are straightforward transcriptions of models from the research literature.

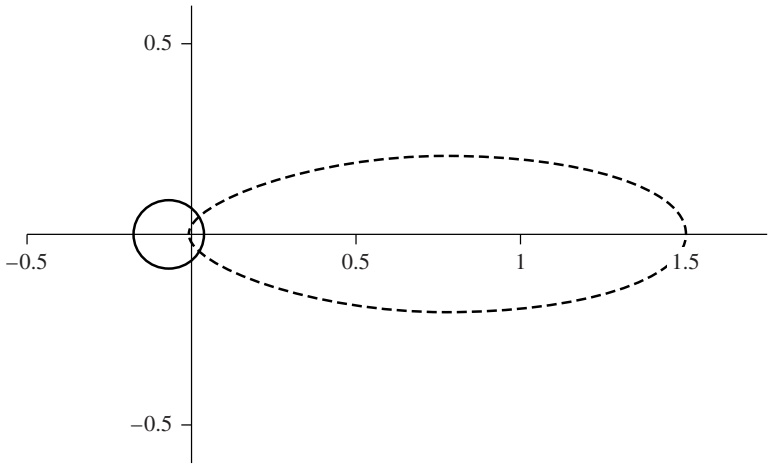


Figure 11.12: Plots of the Henyey–Greenstein Phase Function for Asymmetry g Parameters $-.35$ and $.67$. Negative g values (solid line) describe phase functions that primarily scatter light back in the incident direction, and positive g values (dashed line) describe phase functions that primarily scatter light forward in the direction it was already traveling.

⟨*Volume Scattering Declarations*⟩ ≡
```
float PhaseRayleigh(const Vector &w, const Vector &wp);
float PhaseMieHazy(const Vector &w, const Vector &wp);
float PhaseMieMurky(const Vector &w, const Vector &wp);
```

A widely used phase function, particularly in computer graphics, was developed by Henyey and Greenstein (1941). This phase function was specifically designed to be easy to fit to measured scattering data. A single parameter g (called the *asymmetry parameter*) controls the distribution of scattered light:

$$p_{\mathrm{HG}}(\cos\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g(\cos\theta))^{3/2}}.$$

Figure 11.12 shows plots of the Henyey–Greenstein phase function with varying asymmetry parameters. The value of g for this model must be in the range $(-1, 1)$. Negative values of g correspond to *back-scattering*, where light is mostly scattered back toward the incident direction, and positive values correspond to forward-scattering. The greater the magnitude of g, the more scattering occurs close to the $-\omega$ or ω directions (for back-scattering and forward-scattering, respectively).

⟨*Volume Scattering Definitions*⟩ +≡
```
float PhaseHG(const Vector &w, const Vector &wp, float g) {
    float costheta = Dot(w, wp);
    return 1.f / (4.f * M_PI) *
        (1.f - g*g) / powf(1.f + g*g - 2.f * g * costheta, 1.5f);
}
```

M_PI 1002
Vector 57

The asymmetry parameter has a precise meaning. It is the average value of the product of the phase function being approximated and the cosine of the angle between ω' and ω. Given an arbitrary phase function, g can be computed as

$$g = \int_{S^2} p(\omega \to \omega')(\omega \cdot \omega') \, d\omega' = 2\pi \int_0^\pi p(\cos\theta) \cos\theta \sin\theta \, d\theta.$$

Thus, an isotropic phase function gives $g = 0$, as expected.

Any number of phase functions can satisfy this equation; the g value alone is not enough to uniquely describe a scattering distribution. Nevertheless, the convenience of being able to easily convert a complex scattering distribution into a simple parameterized model is often more important than this potential loss in accuracy.

More complex phase functions that aren't described well with a single asymmetry parameter can often be modeled by a weighted sum of phase functions like Henyey–Greenstein, each with different parameter values:

$$p(\omega \to \omega') = \sum_{i=1}^{n} w_i \, p_i(\omega \to \omega'),$$

where the weights w_i sum to one to maintain normalization.

Blasi, Le Saëc, and Schlick (1993) developed an efficient approximation to the Henyey–Greenstein function that has been widely used in computer graphics due to its computational efficiency. It avoids the expensive `powf()` function:

$$p_{\text{Schlick}}(\cos\theta) = \frac{1}{4\pi} \frac{1 - k^2}{(1 - k\cos\theta)^2}.$$

The k parameter has a similar effect on the distribution to the g term of the Henyey–Greenstein model, where -1 corresponds to total back-scattering, 0 corresponds to isotropic scattering, and 1 corresponds to total forward-scattering. For immediate values, we have found that the polynomial equation

$$k = 1.55g - .55g^3$$

gives an accurate correspondence between k and g values. Figure 11.13 shows a plot of the Schlick model and the Henyey–Greenstein model.

⟨*Volume Scattering Definitions*⟩ +≡

```
float PhaseSchlick(const Vector &w, const Vector &wp, float g) {
    float k = 1.55f * g - .55f * g * g * g;
    float kcostheta = k * Dot(w, wp);
    return 1.f / (4.f * M_PI) *
        (1.f - k*k) / ((1.f - kcostheta) * (1.f - kcostheta));
}
```

M_PI 1002
Vector 57

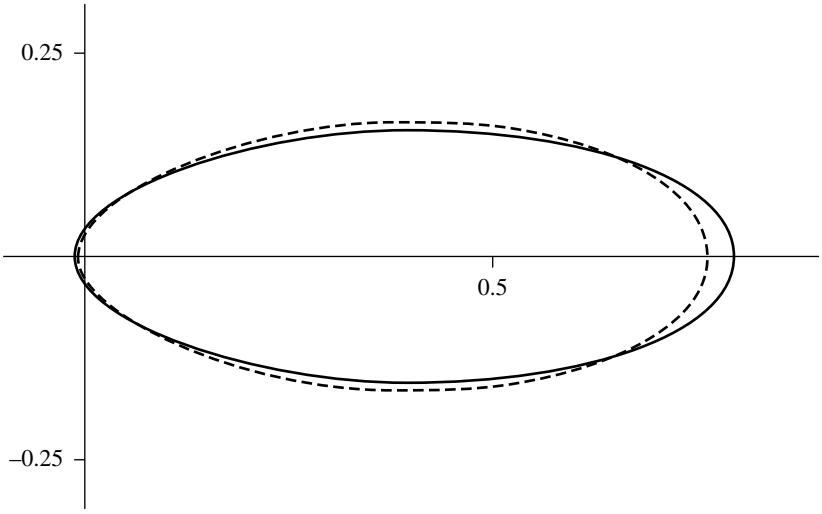


Figure 11.13: Plot of the Schlick phase function with $k = .8112$ (dashed lines) superimposed with plot of the Henyey–Greenstein phase function (solid lines) with $g = .6$. The Schlick model closely matches Henyey–Greenstein and is less computationally expensive to evaluate.

11.3 VOLUME INTERFACE AND HOMOGENEOUS MEDIA

The key abstraction for describing volume scattering in `pbrt` is the abstract `VolumeRegion` class—the interface to describe volume scattering in a region of the scene. Multiple `VolumeRegions` of different types can be used to describe different kinds of scattering in different parts of the scene. In this section, we will describe the basic interface, which is defined in the files `core/volume.h` and `core/volume.cpp`, as well as a few useful implementations of that interface, all of which are in the `volumes/` directory of the `pbrt` distribution.

⟨*Volume Scattering Declarations*⟩ +≡
```
class VolumeRegion {
public:
    ⟨VolumeRegion Interface 587⟩
};
```

All `VolumeRegions` must be able to compute their axis-aligned world space bounding box, which is returned by the `VolumeRegion::WorldBound()` method. As with `Shapes` and `Primitives`, this bound can be used to place `VolumeRegions` into acceleration structures.

BBox 70
VolumeRegion 587
VolumeRegion::WorldBound() 587

⟨*VolumeRegion Interface*⟩ ≡ 587
```
virtual BBox WorldBound() const = 0;
```

Because VolumeIntegrators need to know the parametric range of a world space ray that passes through a volume region, and because a world space bounding box may not tightly bound a particular VolumeRegion, a separate method VolumeRegion::IntersectP() returns the parametric t range of the segment that overlaps the volume, if any.

⟨*VolumeRegion Interface*⟩ +≡ 587
```
virtual bool IntersectP(const Ray &ray, float *t0, float *t1) const = 0;
```

This interface has four methods corresponding to the spatially varying scattering properties introduced earlier in this chapter. Given a world space point, direction, and time, VolumeRegion::sigma_a(), VolumeRegion::sigma_s(), and VolumeRegion::Lve() return the corresponding absorption, scattering, and emission properties. Given a pair of directions, the VolumeRegion::p() method returns the value of the phase function at the given point and the given time.

⟨*VolumeRegion Interface*⟩ +≡ 587
```
virtual Spectrum sigma_a(const Point &, const Vector &,
                         float time) const = 0;
virtual Spectrum sigma_s(const Point &, const Vector &,
                         float time) const = 0;
virtual Spectrum Lve(const Point &, const Vector &,
                     float time) const = 0;
virtual float p(const Point &, const Vector &,
                const Vector &, float time) const = 0;
```

For convenience, there is also a VolumeRegion::sigma_t() method that returns the attenuation coefficient at a point. A default implementation returns the sum of the σ_a and σ_s values, but most of the VolumeRegion implementations will override this method and compute σ_t directly. For VolumeRegions that need to do some amount of computation to find the values of σ_a and σ_s at a particular point, it's usually possible to avoid some duplicated work when σ_t is actually needed and to compute its value directly.

⟨*Volume Scattering Definitions*⟩ +≡
```
Spectrum VolumeRegion::sigma_t(const Point &p, const Vector &w,
                               float time) const {
    return sigma_a(p, w, time) + sigma_s(p, w, time);
}
```

Finally, the VolumeRegion::tau() method computes the volume's optical thickness from the point ray(ray.mint) to ray(ray.maxt). Some implementations, like the HomogeneousVolumeDensity in the next section, can compute this value exactly, while others use Monte Carlo integration to compute it. For the benefit of the Monte Carlo approach, this method takes two optional parameters, step and offset, that are ignored by implementations that compute this value in closed form. These Monte Carlo routines are defined in Section 14.7, and the meanings of these extra parameters are described there.

⟨*VolumeRegion Interface*⟩ +≡ 587
```
virtual Spectrum tau(const Ray &ray, float step = 1.f,
                     float offset = 0.5) const = 0;
```

HomogeneousVolumeDensity 589
Point 63
Ray 66
Spectrum 263
Vector 57
VolumeIntegrator 876
VolumeRegion 587
VolumeRegion::IntersectP() 588
VolumeRegion::Lve() 588
VolumeRegion::p() 588
VolumeRegion::sigma_a() 588
VolumeRegion::sigma_s() 588
VolumeRegion::sigma_t() 588
VolumeRegion::tau() 588



Figure 11.14: Dragon Model Inside a Homogeneous Volume with Uniform Scattering Properties throughout Its Spatial Extent.

11.3.1 HOMOGENEOUS VOLUMES

The simplest volume representation, `HomogeneousVolumeDensity`, describes a box-shaped region of space with homogeneous scattering properties. Values for σ_a, σ_s, the phase function's g value, and the amount of emission L_{ve} are passed to the constructor. In conjunction with a transformation from world space to the volume's object space and an axis-aligned object space bound, this suffices to describe the region's scattering properties and spatial extent. Its implementation is in `volumes/homogeneous.h` and `volumes/homogeneous.cpp`. Figure 11.14 shows an image of the dragon model inside a homogeneous volume.

⟨*HomogeneousVolumeDensity Declarations*⟩ ≡

```
class HomogeneousVolumeDensity : public VolumeRegion {
public:
    ⟨HomogeneousVolumeDensity Public Methods 590⟩
private:
    ⟨HomogeneousVolumeDensity Private Data 590⟩
};
```

HomogeneousVolumeDensity 589
VolumeRegion 587

The constructor, not shown here, initializes the member variables by copying the corresponding parameters.

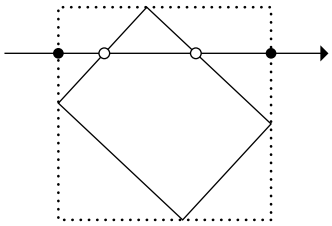


Figure 11.15: Volumes described by axis-aligned bounding boxes in the volume's object space (solid box) can compute a tighter bound on the parametric t range of a ray that overlaps the volume by transforming the ray into object space and computing the ray–box intersections there than if they find the intersections with the world space bound (dotted box) and an untransformed ray in world space. Here, the filled circles denote the world space intersections, and the open circles denote the object space intersections.

⟨*HomogeneousVolumeDensity Private Data*⟩ ≡ 589
```
Spectrum sig_a, sig_s, le;
float g;
BBox extent;
Transform WorldToVolume;
```

Because the bound is maintained internally in the volume's object space, it must be transformed to world space for the `WorldBound()` method.

⟨*HomogeneousVolumeDensity Public Methods*⟩ ≡ 589
```
BBox WorldBound() const {
    return Inverse(WorldToVolume)(extent);
}
```

If the region's world-to-object-space transformation includes a rotation such that the volume isn't axis aligned in world space, it is possible to compute a tighter segment of the ray–volume overlap by transforming the ray to the volume's object space and doing the overlap test there (Figure 11.15).

⟨*HomogeneousVolumeDensity Public Methods*⟩ +≡ 589
```
bool IntersectP(const Ray &r, float *t0, float *t1) const {
    Ray ray = WorldToVolume(r);
    return extent.IntersectP(ray, t0, t1);
}
```

The rest of the `VolumeRegion` interface methods are straightforward. Each one verifies that the given point is inside the region's extent and returns the appropriate value if so. The `sigma_a()` method illustrates the basic approach; the rest of the methods won't be included here.

BBox 70
BBox::IntersectP() 194
HomogeneousVolumeDensity::
extent 590
HomogeneousVolumeDensity::
WorldToVolume 590
Inverse() 1021
Ray 66
Spectrum 263
Transform 76
VolumeRegion 587

⟨*HomogeneousVolumeDensity Public Methods*⟩ +≡ 589
```
Spectrum sigma_a(const Point &p, const Vector &, float) const {
    return extent.Inside(WorldToVolume(p)) ? sig_a : 0.;
}
```

Because σ_a and σ_s are constant throughout the volume, the optical thickness that a ray passes through can be computed in closed form using Beer's law; see Equation (11.1).

⟨*HomogeneousVolumeDensity Public Methods*⟩ +≡ 589
```
Spectrum tau(const Ray &ray, float, float) const {
    float t0, t1;
    if (!IntersectP(ray, &t0, &t1)) return 0.;
    return Distance(ray(t0), ray(t1)) * (sig_a + sig_s);
}
```

11.4 VARYING-DENSITY VOLUMES

The rest of the volume representations in this chapter are based on the assumption that the underlying particles throughout the medium all have the same basic scattering properties, but their density is spatially varying in the medium. One consequence of this assumption is that it is possible to describe the volume scattering properties at a point as the product of the density at that point and some baseline value. For example, we might set the attenuation coefficient σ_t to have a base value of 0.2. In regions where the particle density was 1, a σ_t value of 0.2 would be returned. If the particle density were 3, however, a σ_t value of 0.6 would be the result.

In order to reduce duplicated code and allow the various representations to focus on different methods for defining the density of the particles, we will define a `DensityRegion` class that provides a new pure virtual method to obtain the particle density at a point. Volume representations can then inherit from `DensityRegion` and need not reimplement the substantial amount of shared logic that multiplies the density values by the base values at lookup points.

⟨*Volume Scattering Declarations*⟩ +≡
```
class DensityRegion : public VolumeRegion {
public:
    ⟨DensityRegion Public Methods 592⟩
protected:
    ⟨DensityRegion Protected Data 592⟩
};
```

The `DensityRegion` constructor takes the basic values of the scattering properties and stores them in corresponding member variables. Note that the interface specifies the volume-to-world transformation, but the class instead stores the world-to-volume transformation.

BBox::Inside() 72
DensityRegion 591
Distance() 65
HomogeneousVolumeDensity::extent 590
HomogeneousVolumeDensity::IntersectP() 590
HomogeneousVolumeDensity::sig_a 590
HomogeneousVolumeDensity::sig_s 590
HomogeneousVolumeDensity::WorldToVolume 590
Point 63
Ray 66
Spectrum 263
Vector 57
VolumeRegion 587

⟨*DensityRegion Public Methods*⟩ ≡ 591
```
DensityRegion(const Spectrum &sa, const Spectrum &ss, float gg,
              const Spectrum &emit, const Transform &VolumeToWorld)
    : sig_a(sa), sig_s(ss), le(emit), g(gg),
      WorldToVolume(Inverse(VolumeToWorld)) { }
```

⟨*DensityRegion Protected Data*⟩ ≡ 591
```
Spectrum sig_a, sig_s, le;
float g;
Transform WorldToVolume;
```

All `DensityRegion` implementations must implement the `DensityRegion::Density()` method, which returns the volume's density at the given point in object space. The density is used to scale the basic scattering parameters, so it must be nonnegative everywhere.

⟨*DensityRegion Public Methods*⟩ +≡ 591
```
virtual float Density(const Point &Pobj) const = 0;
```

The `DensityRegion::sigma_a()` method is illustrative of how a `DensityRegion` works; it scales `DensityRegion::sig_a` by the local density at the point. The other `VolumeRegion` methods are similar and not shown here.

⟨*DensityRegion Public Methods*⟩ +≡ 591
```
Spectrum sigma_a(const Point &p, const Vector &, float) const {
    return Density(WorldToVolume(p)) * sig_a;
}
```

One exception is the `DensityRegion::p()` method, which does not scale the phase function's value by the local density. Variations in the amount of scattering from point to point are already accounted for by the scaled σ_s values.

⟨*DensityRegion Public Methods*⟩ +≡ 591
```
float p(const Point &p, const Vector &w, const Vector &wp, float) const {
    return PhaseHG(w, wp, g);
}
```

The `DensityRegion` cannot implement the `VolumeRegion::tau()` method in closed form, since this method depends on global knowledge of the shape of the `VolumeRegion` as well as the density distribution throughout it. However, it can be implemented with Monte Carlo, as is done in Section 15.7.2.

11.4.1 3D GRIDS

The `VolumeGridDensity` class stores densities at a regular 3D grid of positions, similar to the way that the `ImageTexture` represents images with a 2D grid of samples. These samples are interpolated to compute the density at positions between the sample points. The constructor takes a 3D array of user-supplied density values, thus allowing a variety of sources of data (physical simulation, CT scan, etc.). The smoke data set rendered in Figures 11.2, 11.5, and 11.10 is represented with a `VolumeGridDensity`. Because this class is a subclass of `DensityRegion`, the user also supplies baseline values of σ_a,

DensityRegion 591
DensityRegion::Density() 592
DensityRegion::g 592
DensityRegion::p() 592
DensityRegion::sigma_a() 592
DensityRegion::sig_a 592
DensityRegion::WorldToVolume 592
ImageTexture 524
Inverse() 1021
PhaseHG() 585
Point 63
Spectrum 263
Transform 76
Vector 57
VolumeGridDensity 593
VolumeRegion 587
VolumeRegion::tau() 588

σ_s, L_{ve}, and g to the constructor. The implementation of the VolumeGridDensity is in volumes/volumegrid.h and volumes/volumegrid.cpp.

⟨*VolumeGridDensity Declarations*⟩ ≡
```
class VolumeGridDensity : public DensityRegion {
public:
    ⟨VolumeGridDensity Public Methods 593⟩
private:
    ⟨VolumeGridDensity Private Data 593⟩
};
```

The constructor does the usual initialization of the basic scattering properties, stores an object space bounding box for the region, and makes a local copy of the density values.

⟨*VolumeGridDensity Public Methods*⟩ ≡ 593
```
VolumeGridDensity(const Spectrum &sa, const Spectrum &ss, float gg,
        const Spectrum &emit, const BBox &e, const Transform &v2w,
        int x, int y, int z, const float *d)
    : DensityRegion(sa, ss, gg, emit, v2w), nx(x), ny(y), nz(z), extent(e) {
    density = new float[nx*ny*nz];
    memcpy(density, d, nx*ny*nz*sizeof(float));
}
```

⟨*VolumeGridDensity Private Data*⟩ ≡ 593
```
float *density;
const int nx, ny, nz;
const BBox extent;
```

The implementations of the WorldBound() and IntersectP() methods are just like the ones for the HomogeneousVolumeDensity and so aren't included here.

The task of the Density() method of the VolumeGridDensity is to use the samples to reconstruct the volume density function at the given point.

⟨*VolumeGridDensity Method Definitions*⟩ ≡
```
float VolumeGridDensity::Density(const Point &Pobj) const {
    if (!extent.Inside(Pobj)) return 0;
    ⟨Compute voxel coordinates and offsets for Pobj 594⟩
    ⟨Trilinearly interpolate density values to compute local density 594⟩
}
```

Given the eight sample values that surround a point in 3D, this method trilinearly interpolates them to compute the density function's value at the point. It starts by using BBox::Offset() to find the $[0, 1]^3$ offset of the point inside the bounding box and then scaling by (nx, ny, nz) to find the closest volume sample whose integer coordinates are all less than the sample location. It then uses the Manhattan distances along each axis, (dx, dy, and dz), as the interpolants. Note that the same conventions for discrete versus continuous texel coordinates are used here as were used in the ImageFilm and MIPMap (Section 7.1.7).

BBox 70
BBox::Inside() 72
BBox::Offset() 73
DensityRegion 591
HomogeneousVolumeDensity 589
ImageFilm 404
MIPMap 530
Point 63
Spectrum 263
Transform 76
VolumeGridDensity 593
VolumeGridDensity::
 extent 593

⟨*Compute voxel coordinates and offsets for* Pobj⟩ ≡ 593
```
Vector vox = extent.Offset(Pobj);
vox.x = vox.x * nx - .5f;
vox.y = vox.y * ny - .5f;
vox.z = vox.z * nz - .5f;
int vx = Floor2Int(vox.x), vy = Floor2Int(vox.y), vz = Floor2Int(vox.z);
float dx = vox.x - vx, dy = vox.y - vy, dz = vox.z - vz;
```

These distances can be used directly in a series of invocations of `Lerp()` to estimate the density at the sample point:

⟨*Trilinearly interpolate density values to compute local density*⟩ ≡ 593
```
float d00 = Lerp(dx, D(vx, vy, vz),     D(vx+1, vy, vz));
float d10 = Lerp(dx, D(vx, vy+1, vz),   D(vx+1, vy+1, vz));
float d01 = Lerp(dx, D(vx, vy, vz+1),   D(vx+1, vy, vz+1));
float d11 = Lerp(dx, D(vx, vy+1, vz+1), D(vx+1, vy+1, vz+1));
float d0 = Lerp(dy, d00, d10);
float d1 = Lerp(dy, d01, d11);
return Lerp(dz, d0, d1);
```

The `D()` utility method returns the density at the given sample position. Its only tasks are to handle out-of-bounds sample positions with clamping and to compute the appropriate array offset for the given sample. Unlike `MIPMaps`, clamping is almost always the desired solution for out-of-bounds coordinates here. Because all lookup points are inside the `VolumeGridDensity`'s bounding box, the only time we will have out-of-bounds coordinates is at the edges, either due to the offsets for linear interpolation or the continuous-to-discrete texel coordinate conversion. In both of these cases, clamping is the most sensible solution.

⟨*VolumeGridDensity Public Methods*⟩ +≡ 593
```
float D(int x, int y, int z) const {
    x = Clamp(x, 0, nx-1);
    y = Clamp(y, 0, ny-1);
    z = Clamp(z, 0, nz-1);
    return density[z*nx*ny + y*nx + x];
}
```

11.4.2 EXPONENTIAL DENSITY

Another useful density class is `ExponentialDensity`, which describes a density that varies as an exponential function of height h within a given 3D extent:

$$d(h) = ae^{-bh}.$$

The a and b values are parameters that control the overall density and how quickly it falls off as a function of height, respectively. This density function is a good model for the Earth's atmosphere as seen from the Earth's surface, where the atmosphere's curvature can generally be neglected. It can also be used to model low-lying fog at ground level. It is shown in Figure 11.16 and is defined in `volumes/exponential.h` and `volumes/exponential.cpp`.

BBox::Offset() 73
ExponentialDensity 595
Floor2Int() 1002
Lerp() 1000
MIPMap 530
Vector 57
VolumeGridDensity 593
VolumeGridDensity::D() 594
VolumeGridDensity::extent 593
VolumeGridDensity::nx 593
VolumeGridDensity::ny 593
VolumeGridDensity::nz 593



Figure 11.16: `ExponentialDensity` Used in a Scene with the Dragon Model. Note the reduction in density as a function of height. (Compare to Figure 11.14, for example.)

⟨*ExponentialDensity Declarations*⟩ ≡
```
class ExponentialDensity : public DensityRegion {
public:
    ⟨ExponentialDensity Public Methods 595⟩
private:
    ⟨ExponentialDensity Private Data 596⟩
};
```

The `ExponentialDensity` constructor initializes its member variables directly from its arguments. In addition to the volume scattering properties passed to the `DensityRegion` constructor, the volume's bound, and the a and b parameter values, this constructor takes a vector giving an "up" direction that orients the volume and is used to compute the height of points for the density computation. While the up direction is not strictly necessary (the world-to-object transformation is sufficient to orient the volume), specifying an explicit up vector can be conceptually easier for the user.

BBox 70
DensityRegion 591
ExponentialDensity 595
ExponentialDensity::extent 596
ExponentialDensity::upDir 596
Spectrum 263
Transform 76
Vector 57
Vector::Normalize() 63

⟨*ExponentialDensity Public Methods*⟩ ≡ 595
```
ExponentialDensity(const Spectrum &sa, const Spectrum &ss,
                   float gg, const Spectrum &emit, const BBox &e,
                   const Transform &v2w, float aa, float bb,
                   const Vector &up)
    : DensityRegion(sa, ss, gg, emit, v2w), extent(e), a(aa), b(bb) {
    upDir = Normalize(up);
}
```

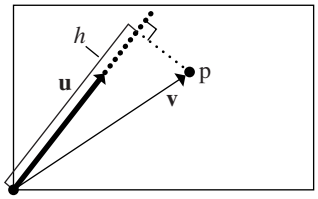


Figure 11.17: For the `ExponentialDensity`, it's necessary to find the perpendicular projection of the point p onto the "up" direction vector **u** and determine the distance h along **u** of the projection point. This distance is given by the dot product $(\mathbf{u} \cdot \mathbf{v})$, where **v** is the vector from the corner of the box (and the up direction's starting point) to p. This can be verified with basic properties of the dot product.

⟨*ExponentialDensity Private Data*⟩ ≡ 595
```
BBox extent;
float a, b;
Vector upDir;
```

The `ExponentialDensity::WorldBound()` and `ExponentialDensity::IntersectP()` methods are the same as their `HomogeneousVolumeDensity` counterparts and are not shown here.

The height of a given object space point p along the "up" direction axis can be found by projecting the vector from the lower corner of the bounding box to p onto the "up" direction vector (Figure 11.17). The distance h along the up direction to the point of p's perpendicular projection is given by the dot product of these two vectors. The principles behind this relationship are similar to those used when vectors are transformed to and from the BSDF coordinate system, for example.

⟨*ExponentialDensity Public Methods*⟩ +≡ 595
```
float Density(const Point &Pobj) const {
    if (!extent.Inside(Pobj)) return 0;
    float height = Dot(Pobj - extent.pMin, upDir);
    return a * expf(-b * height);
}
```

11.5 VOLUME AGGREGATES

Just as `Aggregate` implementations can hold sets of `Primitives`, the `AggregateVolume` holds one or more `VolumeRegions`. There are two main reasons to provide volume aggregates in `pbrt`. First, doing so simplifies the `Scene` and the implementation of `VolumeIntegrators`, since they can both be written to make calls to a single aggregate volume region, rather than looping over all of the regions in the scene. Second, `AggregateVolume` implementations have the potential to use 3D spatial data structures to improve efficiency by culling volumes that are far from a particular ray or lookup point.

Aggregate 192
AggregateVolume 597
BBox 70
ExponentialDensity 595
ExponentialDensity::a 596
ExponentialDensity::b 596
HomogeneousVolumeDensity 589
Point 63
Primitive 185
Scene 22
Vector 57
VolumeIntegrator 876
VolumeRegion 587

Here, we will just implement a simple AggregateVolume that stores an array of all the volumes in the scene and loops over them in each of its method implementations. This implementation is inefficient for scenes with many distinct VolumeRegions. Writing a more efficient implementation is left as an exercise at the end of the chapter.

⟨*Volume Scattering Declarations*⟩ +≡
```
class AggregateVolume : public VolumeRegion {
public:
    ⟨AggregateVolume Public Methods⟩
private:
    ⟨AggregateVolume Private Data 597⟩
};
```

The AggregateVolume constructor is simple. It copies the vector of VolumeRegions passed in and computes the bound that encloses them all.

⟨*Volume Scattering Definitions*⟩ +≡
```
AggregateVolume::AggregateVolume(const vector<VolumeRegion *> &r) {
    regions = r;
    for (uint32_t i = 0; i < regions.size(); ++i)
        bound = Union(bound, regions[i]->WorldBound());
}
```

⟨*AggregateVolume Private Data*⟩ ≡ 597
```
vector<VolumeRegion *> regions;
BBox bound;
```

As described earlier, the implementations of most of the various VolumeRegion interface methods loop over the individual regions. We will show AggregateVolume::sigma_a() as an example; the rest are similar and not shown here. Note that it forwards the call on to each contained VolumeRegion and adds the results. This approach works as desired because the individual VolumeRegion::sigma_a() methods return zero for points outside of their respective extents.

⟨*Volume Scattering Definitions*⟩ +≡
```
Spectrum AggregateVolume::sigma_a(const Point &p, const Vector &w,
                                  float time) const {
    Spectrum s(0.);
    for (uint32_t i = 0; i < regions.size(); ++i)
        s += regions[i]->sigma_a(p, w, time);
    return s;
}
```

AggregateVolume 597
AggregateVolume::regions 597
AggregateVolume::sigma_a() 597
BBox 70
Point 63
Spectrum 263
Vector 57
VolumeRegion 587
VolumeRegion::sigma_a() 588

The one method of this class that isn't completely trivial is IntersectP(). The parametric t range of the ray over all the volumes is equal to the extent from the minimum of all of the regions' t_{min} values to the maximum of all of the t_{max} values.

⟨*Volume Scattering Definitions*⟩ +≡
```
bool AggregateVolume::IntersectP(const Ray &ray,
                                 float *t0, float *t1) const {
    *t0 = INFINITY;
    *t1 = -INFINITY;
    for (uint32_t i = 0; i < regions.size(); ++i) {
        float tr0, tr1;
        if (regions[i]->IntersectP(ray, &tr0, &tr1)) {
            *t0 = min(*t0, tr0);
            *t1 = max(*t1, tr1);
        }
    }
    return (*t0 < *t1);
}
```

11.6 THE BSSRDF

The BSSRDF class plays a role similar to that of the BSDF in pbrt. While the BSDF describes scattering from a surface due to illumination at a single point, the BSSRDF describes the scattering properties for illumination that arrives at many points on the surface and undergoes subsurface scattering before exiting at the point being shaded. During the shading process, BSSRDF pointers are returned by the Primitive::GetBSSRDF() method if the object's material exhibits subsurface scattering.

⟨*BSSRDF Declarations*⟩ ≡
```
class BSSRDF {
public:
    ⟨BSSRDF Public Methods 598⟩
private:
    ⟨BSSRDF Private Data 599⟩
};
```

Unlike the BSDF class, which provides methods to evaluate the value of the BSDF given pairs of angles, the BSSRDF only describes the lower-level scattering properties of the participating medium. It's up to the integrator that computes the effect of subsurface scattering to use these properties in whichever algorithm it uses to compute the value of the BSSRDF for given pairs of incident and exitant locations and directions. (For example, the DipoleSubsurfaceIntegrator of Section 16.5 uses them to compute a BSSRDF approximation based on a closed-form solution for a semi-infinite slab.)

The properties that define the BSSRDF here are the index of refraction of the medium η, its absorption coefficient σ_a, and its *reduced scattering coefficient*, which is defined as $\sigma_s' = (1 - g)\sigma_s$, where g is the medium's anisotropy parameter and σ_s is the scattering coefficient. The reduced scattering coefficient is discussed further in Section 16.5.3.

⟨*BSSRDF Public Methods*⟩ ≡ 598
```
BSSRDF(const Spectrum &sa, const Spectrum &sps, float et)
    : e(et), sig_a(sa), sigp_s(sps) { }
```

AggregateVolume 597
AggregateVolume::regions 597
BSDF 478
BSSRDF 598
DipoleSubsurfaceIntegrator 887
INFINITY 1002
Primitive::GetBSSRDF() 187
Ray 66
Spectrum 263
VolumeRegion::IntersectP() 588

⟨*BSSRDF Private Data*⟩ ≡ 598
```
float e;
Spectrum sig_a, sigp_s;
```

The BSSRDF provides methods to get the values of the three properties. Note that the abstraction here implicitly treats these 3D properties of the medium as properties that only vary on the surface, since the `Material` returns a `BSSRDF`, and the `BSSRDF` doesn't represent any spatial variation of the scattering properties. Effectively, the entire medium is treated as if it were homogeneous, with the scattering properties computed at the point being shaded. This limitation is fine for homogeneous scattering media, but is inaccurate for highly inhomogeneous media.

⟨*BSSRDF Public Methods*⟩ +≡ 598
```
float eta() const { return e; }
Spectrum sigma_a() const { return sig_a; }
Spectrum sigma_prime_s() const { return sigp_s; }
```

11.6.1 SUBSURFACE SCATTERING MATERIALS

There are two `Materials` for translucent objects: `SubsurfaceMaterial`, which is defined in `materials/subsurface.h` and `materials/subsurface.cpp`, and `KdSubsurfaceMaterial`, defined in `materials/kdsubsurface.h` and `materials/kdsubsurface.cpp`. The only difference between these two materials is how the scattering properties of the medium are specified.

⟨*SubsurfaceMaterial Declarations*⟩ ≡
```
class SubsurfaceMaterial : public Material {
public:
    ⟨SubsurfaceMaterial Public Methods⟩
private:
    ⟨SubsurfaceMaterial Private Data 599⟩
};
```

`SubsurfaceMaterial` stores textures that allow the scattering properties to vary as a function of the position on the surface. Note that this isn't the same as scattering properties that vary as a function of 3D inside the scattering medium, but it can give a reasonable approximation to heterogeneous media in some cases.

In addition to the volumetric scattering properties, a specular reflection term `Kr` accounts for specular reflection from incident rays that don't enter the medium due to the different indices of refraction at the boundary. The `SubsurfaceMaterial::GetBSDF()` method adds a `SpecularReflection` component to the `BSDF` if `Kr` isn't black.

⟨*SubsurfaceMaterial Private Data*⟩ ≡ 599
```
float scale;
Reference<Texture<Spectrum> > Kr, sigma_a, sigma_prime_s;
Reference<Texture<float> > eta, bumpMap;
```

The `SubsurfaceMaterial::GetBSSRDF()` method uses the textures to compute the values of the scattering properties at the point. The absorption and reduced scattering coefficients are then scaled by the `scale` member variable, which provides an easy way to

BSDF 478
BSSRDF 598
KdSubsurfaceMaterial 600
Material 483
Reference 1011
Spectrum 263
SpecularReflection 440
SubsurfaceMaterial 599
Texture 519

change the units of the scattering properties. (Recall that they're expected to be specified in terms of reciprocal distance measured in meters, m^{-1}. If the scene isn't modeled in meters, the scattering properties can be adjusted accordingly.)

⟨*SubsurfaceMaterial Method Definitions*⟩ ≡
```
BSSRDF *SubsurfaceMaterial::GetBSSRDF(const DifferentialGeometry &dgGeom,
        const DifferentialGeometry &dgShading, MemoryArena &arena) const {
    float e = eta->Evaluate(dgShading);
    return BSDF_ALLOC(arena, BSSRDF)(scale * sigma_a->Evaluate(dgShading),
        scale * sigma_prime_s->Evaluate(dgShading), e);
}
```

Directly setting the absorption and reduced scattering coefficients to achieve a desired visual look is difficult. The parameters have a nonlinear and unintuitive effect on the result. The `KdSubsurfaceMaterial` allows the user to specify the subsurface scattering properties in terms of the diffuse reflectance of the surface and the mean free path—the average distance light travels in the medium before scattering. It then uses the `SubsurfaceFromDiffuse()` utility function, defined in Section 16.5, to compute the corresponding intrinsic scattering properties. Being able to specify translucent materials in this manner is particularly useful in that it makes it possible to use standard texture maps that might otherwise be used for diffuse reflection to define scattering properties (again with the caveat that varying properties on the surface don't properly correspond to varying properties in the medium).

We won't include the definition of `KdSubsurfaceMaterial` here since its implementation just evaluates `Textures` to compute the diffuse reflection and mean free path values, and calls `SubsurfaceFromDiffuse()` to compute the scattering properties needed by the `BSSRDF`.

Finally, `GetVolumeScatteringProperties()` is a utility function that has a small library of measured scattering data for translucent materials; it returns the corresponding scattering properties if it has an entry for the given name. (For a list of the valid names, see the implementation in `core/volume.cpp`.) The data provided by this function is from papers by Jensen et al. (2001b) and Narasimhan et al. (2006).

⟨*Volume Scattering Declarations*⟩ +≡
```
bool GetVolumeScatteringProperties(const string &name, Spectrum *sigma_a,
                                   Spectrum *sigma_prime_s);
```

BSSRDF 598
DifferentialGeometry 102
MemoryArena 1015
Spectrum 263
SubsurfaceFromDiffuse() 913
SubsurfaceMaterial::eta 599
SubsurfaceMaterial::scale 599
SubsurfaceMaterial::sigma_a 599
SubsurfaceMaterial::sigma_prime_s 599
Texture 519
Texture::Evaluate() 520

FURTHER READING

The books written by van de Hulst (1980) and Preisendorfer (1965, 1976) are excellent introductions to volume light transport. The seminal book by Chandrasekhar (1960) is another excellent resource, although it is mathematically challenging. See also the "Further Reading" section of Chapter 16 for more references to this topic.

The Henyey–Greenstein phase function was originally described by Henyey and Greenstein (1941). Detailed discussion of scattering and phase functions, along with derivations of phase functions that describe scattering from independent spheres, cylinders,

and other simple shapes, can be found in van de Hulst's book (1981). Extensive discussion of the Mie and Rayleigh scattering models is also available there. Hansen and Travis's survey article is also a good introduction to the variety of commonly used phase functions (Hansen and Travis 1974).

Just as procedural modeling of textures is an effective technique for shading surfaces, procedural modeling of volume densities can be used to describe realistic-looking volumetric objects like clouds and smoke. Perlin and Hoffert (1989) described early work in this area, and the book by Ebert et al. (2003) has a number of sections devoted to this topic, including further references. More recently, accurate physical simulation of the dynamics of smoke and fire has led to extremely realistic volume data sets, including the ones used in this chapter; see, for example, Fedkiw, Stam, and Jensen (2001).

In this chapter, we have ignored all issues related to sampling and antialiasing of volume density functions that are represented by samples in a 3D grid, although these issues should be considered, especially in the case of a volume that occupies just a few pixels on the screen. Furthermore, we have used a simple triangle filter to reconstruct densities at intermediate positions, which is suboptimal for the same reasons as the triangle filter is not a high-quality image reconstruction filter. Marschner and Lobb (1994) presented the theory and practice of sampling and reconstruction for 3D data sets, applying ideas similar to those in Chapter 7. See also the paper by Theußl, Hauser, and Gröller (2000) for a comparison of a variety of windowing functions for volume reconstruction with the sinc function and a discussion of how to derive optimal parameters for volume reconstruction filter functions.

Acquiring volumetric scattering properties of real-world objects is particularly difficult, requiring solving the inverse problem of determining the values that lead to the measured result. See Jensen et al. (2001b), Goesele et al. (2004), Narasimhan et al. (2006), and Peers et al. (2006) for recent work on acquiring scattering properties for subsurface scattering. Hawkins et al. (2005) have developed techniques to measure properties of media like smoke, acquiring measurements in real time. Another interesting approach to this problem was introduced by Frisvad et al. (2007), who developed methods to compute these properties from a lower-level characterization of the scattering properties of the medium.

EXERCISES

❶ 11.1 The optical thickness of a ray passing through an `ExponentialDensity` can be computed in closed form, so that the default `tau()` method based on Monte Carlo integration isn't needed. Derive this expression and add an implementation that computes its value to the `ExponentialDensity` class. Test your implementation to ensure that it computes the same results as the Monte Carlo approach. How much does this speed up `pbrt` for scenes that use an instance of the `ExponentialDensity` volume?

`ExponentialDensity` 595

❷ 11.2 Given a one-dimensional volume density that is an arbitrary function of height $f(h)$, the optical distance between any two three-dimensional points can be computed very efficiently if the integral $\int_0^{h'} f(h)\,\mathrm{d}h$ is precomputed and stored

in a table for a set of h' values (Perlin 1985b; Max 1986; Legakis 1998). Work through the mathematics to show the derivation for this approach and implement it in pbrt, either by modifying ExponentialDensity to use such a lookup table or by implementing a new VolumeRegion that takes an arbitrary function or a 1D table of density values. How do the efficiency and accuracy of this approach compare to using the default implementation of DensityRegion::tau()?

● 11.3 The VolumeGridDensity class uses a relatively large amount of memory for complex volume densities. Determine its memory requirements when used for the smoke images in this chapter and modify its implementation to reduce memory use. One approach is to detect regions of space with constant (or relatively constant) density values using an octree data structure and to only refine the octree in regions where the densities are changing. Another possibility is to use less memory to record each density value, for example, by computing the minimum and maximum densities and then using 8 or 16 bits per density value to interpolate between them. What sort of errors appear when either of these approaches is pushed too far?

● 11.4 Modify VolumeGridDensity to use an optimized version of the tau() method instead of relying on the generic Monte Carlo-based implementation of DensityRegion::tau(). Incorporate the available information about the distribution of densities to compute the integral using fewer samples in areas where the density is low. For inspiration, see, for example, Levoy (1988), which describes using a binary volume octree to speed up traversal of empty regions, and Danskin and Hanrahan (1992), which describes various techniques based on a 3D pyramid of volume data to use lower-precision computations in regions that have a low contribution to the final result.

● 11.5 The AggregateVolume will have poor performance for a scene with more than a handful of VolumeRegions. For example, time will be wasted determining the values of σ_t and σ_s in areas where the point is outside most of the volumes. Write a better volume aggregate based on a 3D data structure like a grid or an octree. Verify that it returns the same results as the AggregateVolume and measure how much faster pbrt is when it is used instead of AggregateVolume. (Don't forget to modify the RenderOptions::MakeScene() method to create an instance of your new aggregate instead of an AggregateVolume.)

● 11.6 Implement the atmospheric scattering model described by Preetham, Shirley, and Smits (1999) in pbrt. Use it to render images of the ecosystem scene and other outdoor scenes.

AggregateVolume 597
DensityRegion::tau() 733
ExponentialDensity 595
RenderOptions::MakeScene() 1072
VolumeGridDensity 593
VolumeRegion 587

This page intentionally left blank



CHAPTER TWELVE

12 LIGHT SOURCES

In order for objects in a scene to be visible, some of them must emit light that is eventually reflected back to the camera. This chapter introduces the abstract `Light` class, which defines the interface used for light sources in `pbrt`, as well as the implementations of a number of useful light sources. By hiding the implementation of different types of lights behind a carefully designed interface, the light transport routines can operate without knowing which particular types of lights are in the scene, similar to how the acceleration structures can hold collections of primitives without needing to know their actual types. This chapter only defines the basic light functionality because many of the quantities related to complex light sources cannot be computed in closed form. The Monte Carlo routines in Chapter 14 will complete the lighting interface and implementations by introducing methods for numerically approximating these quantities.

A range of light source implementations are introduced in this chapter, although the variety is slightly limited by `pbrt`'s physically based design. Many flexible light source models have been developed for computer graphics, incorporating control over properties like the rate at which the light falls off with distance, which objects are illuminated by the light, which objects cast shadows from the light, and so on. While controls such as these are quite useful for artistic effects, many of them are incompatible with physically based light transport algorithms and thus can't be provided in the models here. As an example of this issue, consider a light that doesn't cast shadows: the total energy arriving at surfaces in the scene increases without bound as more surfaces are added. Consider a series of concentric shells of spheres around such a light; if occlusion is ignored, each added shell increases the total received power. This directly violates the principle that the total energy arriving at surfaces illuminated by the light can't be greater than the total energy emitted by the light.

12.1 LIGHT INTERFACE

The core lighting routines and interfaces are in `core/light.h` and `core/light.cpp`. Implementations of particular lights are in individual source files in the `lights/` directory. All lights share two common parameters: a transformation that defines the light's coordinate system in world space and a parameter that affects Monte Carlo sampling of lights, `nSamples`. As with shapes, it's often handy to be able to implement a light assuming a particular coordinate system (e.g., that a spotlight is always located at the origin of its light space, shining down the $+z$ axis). The light-to-world transformation makes it possible to place such lights at arbitrary positions and orientations in the scene. The `nSamples` parameter is used for area light sources where it may be desirable to trace multiple shadow rays to the light to compute soft shadows; it allows the user to have finer-grained control of the number of samples taken on a per-light basis.

⟨*Light Declarations*⟩ ≡
```
class Light {
public:
    ⟨Light Interface 606⟩
    ⟨Light Public Data 606⟩
protected:
    ⟨Light Protected Data 606⟩
};
```

Although storing both the light-to-world and the world-to-light transformations is redundant, having both available simplifies code elsewhere by eliminating the need for calls for `Inverse()`. The default number of light source samples taken is one; thus, only the light implementations for which taking multiple samples is sensible need to pass in an explicit value here. The only other job for the constructor is to warn if the light-to-world transformation has a scale factor; many of the `Light` methods will return incorrect results in this case.[1]

⟨*Light Interface*⟩ ≡ 606
```
Light(const Transform &l2w, int ns = 1)
    : nSamples(max(1, ns)), LightToWorld(l2w),
      WorldToLight(Inverse(l2w)) {
    ⟨Warn if light has transformation with scale⟩
}
```

⟨*Light Protected Data*⟩ ≡ 606
```
const Transform LightToWorld, WorldToLight;
```

⟨*Light Public Data*⟩ ≡ 606
```
const int nSamples;
```

`Inverse()` 1021
`Light` 606
`Light::LightToWorld` 606
`Light::nSamples` 606
`Light::WorldToLight` 606
`Transform` 76

1 For example, the surface area reported by area lights is computed from the untransformed geometry, so a scale factor in the transformation means that the reported area and the actual area of the light in the scene would be inconsistent.

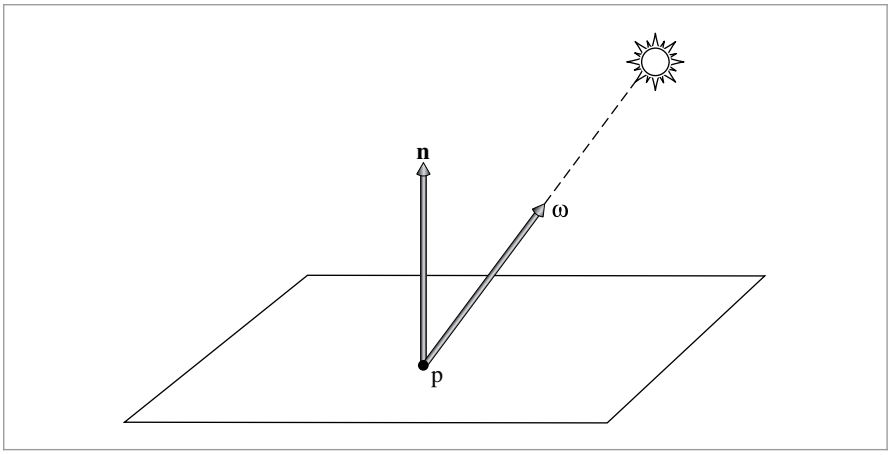


Figure 12.1: The `Light::Sample_L()` method returns incident radiance from the light at a point p and also returns the direction vector ω that gives the direction from which radiance is arriving.

The key method for lights to implement is `Sample_L()`. The caller passes the world space position of a point in the scene and the time at which the light sample is to be taken, and the light returns the radiance arriving at that point due to that light, assuming there are no occluding objects between them (Figure 12.1). Although this method also takes a time value, the `Light` implementations do not currently support being animated themselves—the lights themselves are at fixed positions in the scene. (Addressing this limitation is left as an exercise.) However, the time value is needed to set the corresponding value appropriately in the traced visibility ray.

The light is also responsible for initializing the incident direction to the light source ω_i and for initializing the `VisibilityTester` object, which holds information about the shadow ray that must be traced to verify that there are no occluding objects between the light and p. The `VisibilityTester`, which will be described in Section 12.1.1, need not be initialized if the returned radiance value is black—for example, due to the point p being outside of the cone of illumination of a spotlight. In addition to the point p being illuminated, this method takes an epsilon value for shadow rays leaving p, `pEpsilon`.

For some types of lights, light may arrive at p from many directions, not just from a single direction as with a point light source, for example. For these types of light sources, the `Sample_L()` method must randomly sample a point on the light source's surface, so that Monte Carlo integration can be used to find the reflected light at p due to illumination from the light. Implementations of the `Sample_L()` interface for such lights will be introduced later, in Section 14.6. The `LightSample` is used by these methods, and the `pdf` output parameter stores the probability density for the light sample taken. For all of the implementations in this chapter, the `LightSample` is ignored and the `pdf` is set to one. The `pdf` value's role in the context of Monte Carlo sampling is discussed in Section 14.6.1.

`Light` 606
`Light::Sample_L()` 608
`LightSample` 710
`VisibilityTester` 608

⟨*Light Interface*⟩ +≡ 606
```
virtual Spectrum Sample_L(const Point &p, float pEpsilon,
    const LightSample &ls, float time, Vector *wi, float *pdf,
    VisibilityTester *vis) const = 0;
```

All lights must also be able to return their total emitted power; this quantity is useful for light transport algorithms that may want to devote additional computational resources to lights in the scene that make the largest contribution. Because a precise value for emitted power isn't needed elsewhere in the system, a number of the implementations of this method later in this chapter will compute approximations to this value rather than expending computational effort to find a precise value.

⟨*Light Interface*⟩ +≡ 606
```
virtual Spectrum Power(const Scene *) const = 0;
```

Finally, the `IsDeltaLight()` method indicates whether the light is described by a delta distribution. Examples of such lights include point lights, which emit illumination from a single point, and directional lights, where all light arrives from the same direction. The only way to detect illumination from light sources like these is to call their `Sample_L()` methods. It's impossible to randomly choose a direction from a point p that happens to find such a light source. (This is analogous to delta components in BSDFs from specular reflection or transmission.) The Monte Carlo algorithms that sample illumination from light sources need to be aware of which lights are described by delta distributions, since this affects some of their computations.

⟨*Light Interface*⟩ +≡ 606
```
virtual bool IsDeltaLight() const = 0;
```

12.1.1 VISIBILITY TESTING

The `VisibilityTester` is a *closure*—an object that encapsulates a small amount of data and some computation that is yet to be done. It allows lights to return a radiance value under the assumption that the receiving point p and the light source are mutually visible. The integrator can then decide if illumination from the direction ω is relevant before incurring the cost of tracing the shadow ray—for example, light incident on the back side of a surface that isn't translucent contributes nothing to reflection from the other side. If the actual amount of arriving illumination is in fact needed, a call to one of the visibility tester's methods causes the necessary shadow ray to be traced.

⟨*Light Declarations*⟩ +≡
```
struct VisibilityTester {
    ⟨VisibilityTester Public Methods 609⟩
    Ray r;
};
```

There are two methods that initialize `VisibilityTesters`. The first of them, `VisibilityTester::SetSegment()`, indicates that the visibility test is to be done between two points in the scene. In addition to the two points, it takes "epsilon" values to offset the ray's starting point and ending point, respectively.

LightSample 710
Point 63
Ray 66
Scene 22
Spectrum 263
Vector 57
VisibilityTester 608
VisibilityTester::SetSegment() 609

⟨*VisibilityTester Public Methods*⟩ ≡ 608
```
void SetSegment(const Point &p1, float eps1,
                const Point &p2, float eps2, float time) {
    float dist = Distance(p1, p2);
    r = Ray(p1, (p2-p1) / dist, eps1, dist * (1.f - eps2), time);
}
```

The other initialization method, `VisibilityTester::SetRay()`, indicates that the test should indicate whether there is *any* object along a given direction. This variant is useful for computing shadows from directional lights.

⟨*VisibilityTester Public Methods*⟩ +≡ 608
```
void SetRay(const Point &p, float eps, const Vector &w, float time) {
    r = Ray(p, w, eps, INFINITY, time);
}
```

The next pair of methods trace the appropriate ray. The first one, `VisibilityTester::Unoccluded()` traces the shadow ray and returns a Boolean result. Some ray tracers include a facility for casting colored shadows from partially transparent objects and would return a spectrum from a method like this. `pbrt` does not include this facility explicitly, since those systems typically implement it with a nonphysical hack. Thus, the `VisibilityTester::Unoccluded()` method here returns a Boolean. Scenes where illumination passes through a transparent object should be rendered with an integrator that supports this kind of effect, like the `PhotonIntegrator` described in Section 15.6.

⟨*Light Method Definitions*⟩ ≡
```
bool VisibilityTester::Unoccluded(const Scene *scene) const {
    return !scene->IntersectP(r);
}
```

`VisibilityTester::Transmittance()` determines the fraction of illumination from the light to the point that is not extinguished by participating media in the scene. If the scene has no participating media, this method returns a constant spectral value of one.

⟨*Light Method Definitions*⟩ +≡
```
Spectrum VisibilityTester::Transmittance(const Scene *scene,
        const Renderer *renderer, const Sample *sample,
        RNG &rng, MemoryArena &arena) const {
    return renderer->Transmittance(scene, RayDifferential(r), sample,
                                   rng, arena);
}
```

INFINITY 1002
MemoryArena 1015
PhotonIntegrator 802
Point 63
PointLight 610
Ray 66
RayDifferential 69
Renderer 24
Renderer::Transmittance() 25
RNG 1003
Sample 343
Scene 22
Scene::IntersectP() 24
Spectrum 263
Vector 57
VisibilityTester 608
VisibilityTester::r 608
VisibilityTester::SetRay() 609
VisibilityTester::Unoccluded() 609

12.2 POINT LIGHTS

A number of interesting lights can be described in terms of emission from a single point in space with some possibly angularly varying distribution of outgoing light. This section describes the implementation of a number of them, starting with `PointLight`, which represents an isotropic point light source that emits the same amount of light in all directions. It is defined in `lights/point.h` and `lights/point.cpp`. Figure 12.2 shows

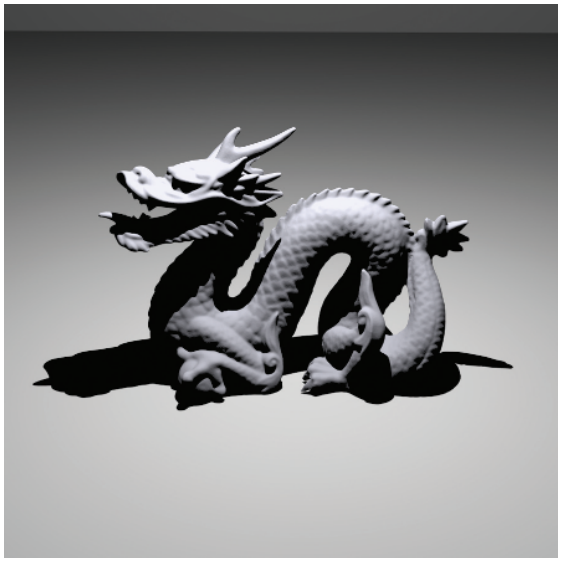


Figure 12.2: Scene Rendered with a Point Light Source. Notice the hard shadow boundaries from this type of light.

a scene rendered with a point light source. Building on this base, a number of more complex lights based on point sources will be introduced, including spotlights and a light that projects an image into the scene.

⟨*PointLight Declarations*⟩ ≡
```
class PointLight : public Light {
public:
    ⟨PointLight Public Methods 611⟩
private:
    ⟨PointLight Private Data 611⟩
};
```

PointLights are positioned at the origin in light space. To place them elsewhere, the light-to-world transformation should be modified as appropriate. Using this transformation, the world space position of the light is precomputed and cached in the constructor by transforming (0, 0, 0) from light space to world space. The constructor also stores the intensity for the light source, which is the amount of power per unit solid angle. Because the light source is isotropic, this is a constant.

⟨*PointLight Method Definitions*⟩ ≡
```
PointLight::PointLight(const Transform &light2world,
                       const Spectrum &intensity)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
}
```

Light 606
Light::LightToWorld 606
Point 63
PointLight 610
PointLight::Intensity 611
PointLight::lightPos 611
Spectrum 263
Transform 76

⟨*PointLight Private Data*⟩ ≡ 610
```
Point lightPos;
Spectrum Intensity;
```

Strictly speaking, it is incorrect to describe the light arriving at a point due to a point light source using units of radiance. Radiant intensity is instead the proper unit for describing emission from a point light source, as explained in Section 5.4. In the light source interfaces here, however, we will abuse terminology and use `Sample_L()` methods to report the illumination arriving at a point for all types of light sources, dividing radiant intensity by the squared distance to the point p to convert units. Section 14.6 revisits the details of this issue in its discussion of how delta distributions affect evaluation of the integral in the scattering equation. In the end, the correctness of the computation does not suffer from this fudge, and it makes the implementation of light transport algorithms more straightforward by not requiring them to use different interfaces for different types of light.

⟨*PointLight Method Definitions*⟩ +≡
```
Spectrum PointLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    *wi = Normalize(lightPos - p);
    *pdf = 1.f;
    visibility->SetSegment(p, pEpsilon, lightPos, 0., time);
    return Intensity / DistanceSquared(lightPos, p);
}
```

The total power emitted by the light source can be found by integrating the intensity over the entire sphere of directions:

$$\Phi = \int_{S^2} I \, d\omega = I \int_{S^2} d\omega = 4\pi I .$$

⟨*PointLight Method Definitions*⟩ +≡
```
Spectrum PointLight::Power(const Scene *) const {
    return 4.f * M_PI * Intensity;
}
```

Finally, since point lights represent singularities that only cast incident light along a single direction, `PointLight::IsDeltaLight()` returns `true`.

⟨*PointLight Public Methods*⟩ ≡ 610
```
bool IsDeltaLight() const { return true; }
```

DistanceSquared() 65
LightSample 710
M_PI 1002
Point 63
PointLight 610
PointLight::Intensity 611
PointLight::IsDeltaLight() 611
PointLight::lightPos 611
Scene 22
Spectrum 263
Vector 57
Vector::Normalize() 63
VisibilityTester 608
VisibilityTester::SetSegment() 609

12.2.1 SPOTLIGHTS

Spotlights are a handy variation on point lights; rather than shining illumination in all directions, they emit light in a cone of directions from their position. For simplicity, we will define the spotlight in the light coordinate system to always be at position (0, 0, 0) and pointing down the $+z$ axis. To place or orient it elsewhere in the scene, the



Figure 12.3: Scene Rendered with a Spotlight. The spotlight cone smoothly cuts off illumination past a user-specified angle from the light's central axis.

`Light::WorldToLight` transformation should be set accordingly. Figure 12.3 shows a rendering of the same scene as Figure 12.2, only illuminated with a spotlight instead of a point light. The `SpotLight` class is defined in `lights/spot.h` and `lights/spot.cpp`.

⟨*SpotLight Declarations*⟩ ≡
```
class SpotLight : public Light {
public:
    ⟨SpotLight Public Methods⟩
private:
    ⟨SpotLight Private Data 613⟩
};
```

Two angles are passed to the constructor to set the extent of the `SpotLight`'s cone: the overall angular width of the cone and the angle at which falloff starts (Figure 12.4). The constructor precomputes and stores the cosines of these angles for use in the `SpotLight`'s methods.

⟨*SpotLight Method Definitions*⟩ ≡
```
SpotLight::SpotLight(const Transform &light2world,
                     const Spectrum &intensity, float width, float fall)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    cosTotalWidth = cosf(Radians(width));
    cosFalloffStart = cosf(Radians(fall));
}
```

Light 606
Light::LightToWorld 606
Light::WorldToLight 606
Point 63
Radians() 1001
Spectrum 263
SpotLight 612
SpotLight::
cosFalloffStart 613
SpotLight::cosTotalWidth 613
SpotLight::Intensity 613
SpotLight::lightPos 613
Transform 76

⟨*SpotLight Private Data*⟩ ≡ 612
```
Point lightPos;
Spectrum Intensity;
float cosTotalWidth, cosFalloffStart;
```

The `SpotLight::Sample_L()` method is almost identical to `PointLight::Sample_L()`, except that it also calls the `Falloff()` method, which computes the distribution of light accounting for the spotlight cone. This computation is encapsulated in a separate method since other `SpotLight` methods will need to perform it as well.

⟨*SpotLight Method Definitions*⟩ +≡
```
Spectrum SpotLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi,
        float *pdf, VisibilityTester *visibility) const {
    *wi = Normalize(lightPos - p);
    *pdf = 1.f;
    visibility->SetSegment(p, pEpsilon, lightPos, 0., time);
    return Intensity * Falloff(-*wi) / DistanceSquared(lightPos, p);
}
```

To compute the spotlight's strength for a receiving point p, this first step is to compute the cosine of the angle between the vector from the spotlight origin to p and the vector along the center of the spotlight's cone. To compute the cosine of the offset angle to a point p, we have (Figure 12.4)

$$\begin{aligned}\cos\theta &= \left(\mathrm{p} \mathbin{\widehat{-}} (0,0,0)\right) \cdot (0,0,1) \\ &= \mathrm{p}_z / \|\mathrm{p}\|.\end{aligned}$$

This value is then compared to the cosines of the falloff and overall width angles to see where the point lies with respect to the spotlight cone. We can trivially determine that points with a cosine greater than the cosine of the falloff angle are inside the cone

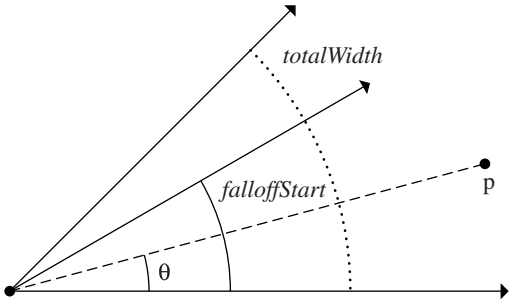


Figure 12.4: Spotlights are defined by two angles, *falloffStart* and *totalWidth*. Objects inside the inner cone of angles, up to *falloffStart*, are fully illuminated by the light. The directions between *falloffStart* and *totalWidth* are a transition zone that ramps down from full illumination to no illumination, such that points outside the *totalWidth* cone aren't illuminated at all. The cosine of the angle between the vector to a point p and the spotlight axis, θ, can easily be computed with a dot product.

DistanceSquared() 65
LightSample 710
Point 63
PointLight::Sample_L() 611
Spectrum 263
SpotLight 612
SpotLight::Falloff() 614
SpotLight::Intensity 613
SpotLight::lightPos 613
SpotLight::Sample_L() 713
Vector 57
Vector::Normalize() 63
VisibilityTester 608
VisibilityTester::
SetSegment() 609

receiving full illumination, and points with cosine less than the width angle's cosine are completely outside the cone. (Note that the computation is slightly tricky since for $\theta \in [0, 2\pi]$, if $\theta > \theta'$, then $\cos\theta < \cos\theta'$.)

⟨*SpotLight Method Definitions*⟩ +≡
```
float SpotLight::Falloff(const Vector &w) const {
    Vector wl = Normalize(WorldToLight(w));
    float costheta = wl.z;
    if (costheta < cosTotalWidth)     return 0.;
    if (costheta > cosFalloffStart)   return 1.;
    ⟨Compute falloff inside spotlight cone 614⟩
}
```

For points inside the transition range from fully illuminated to outside of the cone, the intensity is scaled to smoothly fall off from full illumination to darkness:

⟨*Compute falloff inside spotlight cone*⟩ ≡ 614
```
float delta = (costheta - cosTotalWidth) /
              (cosFalloffStart - cosTotalWidth);
return delta*delta*delta*delta;
```

The solid angle subtended by a cone with spread angle θ is $2\pi(1 - \cos\theta)$. Therefore, the integral over directions on the sphere that gives power from radiant intensity can be solved to compute the total power of a light that only emits illumination in a cone. For the spotlight, we can reasonably approximate the power of the light by computing the solid angle of directions that is covered by the cone with a spread angle cosine halfway between `width` and `fall`.

⟨*SpotLight Method Definitions*⟩ +≡
```
Spectrum SpotLight::Power(const Scene *) const {
    return Intensity * 2.f * M_PI *
           (1.f - .5f * (cosFalloffStart + cosTotalWidth));
}
```

12.2.2 TEXTURE PROJECTION LIGHTS

Another useful light source acts like a slide projector; it takes an image map and projects its image out into the scene. The `ProjectionLight` class uses a projective transformation to project points in the scene onto the light's projection plane based on the field of view angle given to the constructor (Figure 12.5). Its implementation is in `lights/projection.h` and `lights/projection.cpp`. The use of this light in the lighting example scene is shown in Figure 12.6.

⟨*ProjectionLight Declarations*⟩ ≡
```
class ProjectionLight : public Light {
public:
    ⟨ProjectionLight Public Methods⟩
private:
    ⟨ProjectionLight Private Data 616⟩
};
```

Light 606
Light::WorldToLight 606
M_PI 1002
ProjectionLight 614
Scene 22
Spectrum 263
SpotLight 612
SpotLight::cosFalloffStart 613
SpotLight::cosTotalWidth 613
SpotLight::Intensity 613
Vector 57
Vector::Normalize() 63

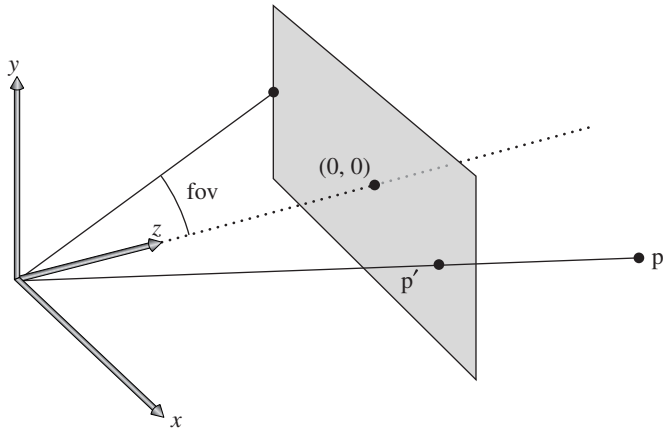


Figure 12.5: The Basic Setting for Projection Light Sources. A point p in the light's coordinate system is projected onto the plane of the image using the light's projection matrix.

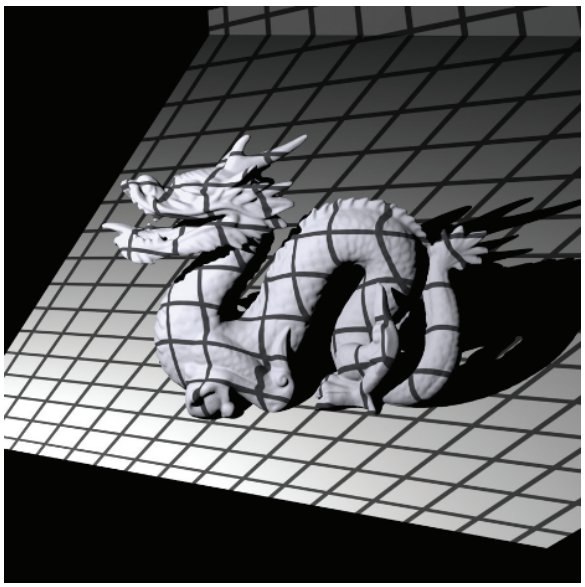


Figure 12.6: Scene Rendered with a Projection Light Using a Grid Texture Map. The projection light acts like a slide projector, projecting an image onto objects in the scene.

⟨*ProjectionLight Method Definitions*⟩ ≡
```
ProjectionLight::ProjectionLight(const Transform &light2world,
        const Spectrum &intensity, const string &texname,
        float fov)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    ⟨Create ProjectionLight MIP-map 616⟩
    ⟨Initialize ProjectionLight projection matrix 616⟩
    ⟨Compute cosine of cone surrounding projection directions 617⟩
}
```

This light could use a `Texture` to represent the light projection distribution so that procedural projection patterns could be used. However, having a precise representation of the projection function, as is available by using an image in a `MIPMap`, is useful for being able to sample the projection distribution using Monte Carlo techniques, so we will use that representation in the implementation here. Note that the projected image is explicitly stored as a `RGBSpectrum`, even if full-spectral rendering is being performed. Unless the image map being used has full spectral data, storing full `SampledSpectrum` values in this case only wastes memory; whether an RGB color is converted to a `SampledSpectrum` before the `MIPMap` is created or after the `MIPMap` returns a value from its `Lookup()` routine gives the same result in either case.

⟨*Create* `ProjectionLight` *MIP-map*⟩ ≡ 616
```
int width, height;
RGBSpectrum *texels = ReadImage(texname, &width, &height);
if (texels)
    projectionMap = new MIPMap<RGBSpectrum>(width, height, texels);
else
    projectionMap = NULL;
delete[] texels;
```

⟨*ProjectionLight Private Data*⟩ ≡ 614
```
MIPMap<RGBSpectrum> *projectionMap;
Point lightPos;
Spectrum Intensity;
```

Similar to the `PerspectiveCamera`, the `ProjectionLight` constructor computes a projection matrix and the screen space extent of the projection.

⟨*Initialize* `ProjectionLight` *projection matrix*⟩ ≡ 616
```
float aspect = float(width) / float(height);
if (aspect > 1.f)  {
    screenX0 = -aspect; screenX1 = aspect;
    screenY0 = -1.f;    screenY1 = 1.f;
}
else {
    screenX0 = -1.f;            screenX1 = 1.f;
    screenY0 = -1.f / aspect;   screenY1 = 1.f / aspect;
}
```

Light 606
Light::LightToWorld 606
MIPMap 530
Perspective() 311
PerspectiveCamera 310
Point 63
ProjectionLight 614
ProjectionLight::hither 617
ProjectionLight::Intensity 616
ProjectionLight::lightPos 616
ProjectionLight::lightProjection 617
ProjectionLight::projectionMap 616
ProjectionLight::screenX0 617
ProjectionLight::screenX1 617
ProjectionLight::screenY0 617
ProjectionLight::screenY1 617
ProjectionLight::yon 617
ReadImage() 1004
RGBSpectrum 279
SampledSpectrum 266
Spectrum 263
Texture 519
Transform 76

```
hither = 1e-3f;
yon = 1e30f;
lightProjection = Perspective(fov, hither, yon);
```

⟨*ProjectionLight Private Data*⟩ +≡ 614
```
Transform lightProjection;
float hither, yon;
float screenX0, screenX1, screenY0, screenY1;
```

Finally, the constructor finds the cosine of the angle between the $+z$ axis and the vector to a corner of the screen window. This value is used elsewhere to define the minimal cone of directions that encompasses the set of directions in which light is projected. This cone is useful for algorithms like photon mapping that need to randomly sample rays leaving the light source. We won't derive this computation here; it is based on straightforward trigonometry.

⟨*Compute cosine of cone surrounding projection directions*⟩ ≡ 616
```
float opposite = tanf(Radians(fov) / 2.f);
float tanDiag = opposite * sqrtf(1.f + 1.f/(aspect*aspect));
cosTotalWidth = cosf(atanf(tanDiag));
```

⟨*ProjectionLight Private Data*⟩ +≡ 614
```
float cosTotalWidth;
```

Similar to the spotlight's version, `ProjectionLight::Sample_L()` calls a utility method, `ProjectionLight::Projection()`, to determine how much light is projected in the given direction. Therefore, we won't include the implementation of `Sample_L()` here.

⟨*ProjectionLight Method Definitions*⟩ +≡
```
Spectrum ProjectionLight::Projection(const Vector &w) const {
    Vector wl = WorldToLight(w);
    ⟨Discard directions behind projection light 617⟩
    ⟨Project point onto projection plane and compute light 618⟩
}
```

Because the projective transformation has the property that it projects points behind the center of projection to points in front of it, it is important to discard points with a negative z value. Therefore, the projection code immediately returns no illumination for projection points that are behind the hither plane for the projection. If this check were not done, then it wouldn't be possible to know if a projected point was originally behind the light (and therefore not illuminated) or in front of it.

⟨*Discard directions behind projection light*⟩ ≡ 617
```
if (wl.z < hither) return 0.;
```

After being projected to the projection plane, points with coordinate values outside the screen window are discarded. Points that pass this test are transformed to get (s, t) texture coordinates inside $[0, 1]^2$ for the lookup in the projection's image map.

Light::WorldToLight 606
ProjectionLight 614
ProjectionLight::
cosTotalWidth 617
ProjectionLight::hither 617
ProjectionLight::
Projection() 617
Radians() 1001
Spectrum 263
Transform 76
Vector 57

⟨*Project point onto projection plane and compute light*⟩ ≡ 617
```
Point Pl = lightProjection(Point(wl.x, wl.y, wl.z));
if (Pl.x < screenX0 || Pl.x > screenX1 ||
    Pl.y < screenY0 || Pl.y > screenY1) return 0.;
if (!projectionMap) return 1;
float s = (Pl.x - screenX0) / (screenX1 - screenX0);
float t = (Pl.y - screenY0) / (screenY1 - screenY0);
return Spectrum(projectionMap->Lookup(s, t), SPECTRUM_ILLUMINANT);
```

The total power of this light is approximated as a spotlight that subtends the same angle as the diagonal of the projected image, scaled by the average intensity in the image map. This approximation becomes increasingly inaccurate as the projected image's aspect ratio becomes less square, for example, and it doesn't account for the fact that texels toward the edges of the image map subtend a larger solid angle than texels in the middle when projected with a perspective projection. Nevertheless, it's a reasonable first-order approximation. Note it is explicitly specified that the `RGBSpectrum` value passed to the `Spectrum` constructor represents an illuminant's SPD and not that of a reflectance. (Recall from Section 5.2.2 that different matching functions are used for converting from RGB to SPDs for illuminants versus reflectances.)

⟨*ProjectionLight Method Definitions*⟩ +≡
```
Spectrum ProjectionLight::Power(const Scene *) const {
    return Spectrum(projectionMap->Lookup(.5f, .5f, .5f),
                    SPECTRUM_ILLUMINANT) *
        Intensity * 2.f * M_PI * (1.f - cosTotalWidth);
}
```

Light 606
MIPMap::Lookup() 540
M_PI 1002
Point 63
ProjectionLight::cosTotalWidth 617
ProjectionLight::Intensity 616
ProjectionLight::lightProjection 617
ProjectionLight::projectionMap 616
ProjectionLight::screenX0 617
ProjectionLight::screenX1 617
ProjectionLight::screenY0 617
ProjectionLight::screenY1 617
RGBSpectrum 279
Scene 22
Spectrum 263
SPECTRUM_ILLUMINANT 277

12.2.3 GONIOPHOTOMETRIC DIAGRAM LIGHTS

A *goniophotometric diagram* describes the angular distribution of luminance from a point light source; it is widely used in illumination engineering to characterize lights. Figure 12.7 shows an example of a goniophotometric diagram in two dimensions. In this section, we'll implement a light source that uses goniophotometric diagrams encoded in 2D image maps to describe the emission distribution of the light. The implementation is very similar to the point light sources defined previously in this section; it scales the intensity based on the outgoing direction according to the goniophotometric diagram's values. Figure 12.8 shows a few goniophotometric diagrams encoded as image maps, and Figure 12.9 shows a scene rendered with a light source that uses one of these images to modulate its directional distribution of illumination.

⟨*GonioPhotometricLight Declarations*⟩ ≡
```
class GonioPhotometricLight : public Light {
public:
    ⟨GonioPhotometricLight Public Methods 621⟩
private:
    ⟨GonioPhotometricLight Private Data 620⟩
};
```

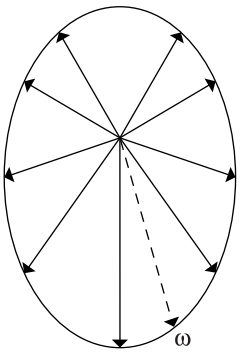


Figure 12.7: An Example of a Goniophotometric Diagram Specifying an Outgoing Light Distribution from a Point Light Source in 2D. The emitted intensity is defined in a fixed set of directions on the unit sphere, and the intensity for a given outgoing direction ω is found by interpolating the intensities of the adjacent samples.

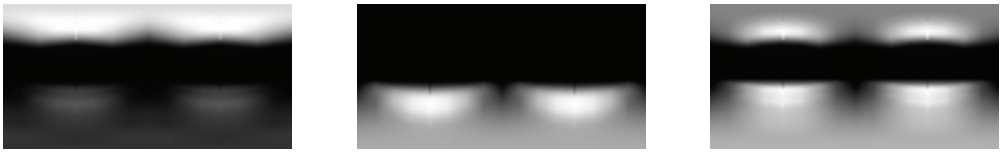


Figure 12.8: Goniophotometric Diagrams for Real-World Light Sources, Encoded as Image Maps with a Parameterization Based on Spherical Coordinates. (a) A light that mostly illuminates in its up direction, with only a small amount of illumination in the down direction. (b) A light that mostly illuminates in the down direction. (c) A light that casts illumination both above and below.

The `GonioPhotometricLight` constructor takes a base intensity and an image map that scales the intensity based on the angular distribution of light.

GonioPhotometricLight 618
GonioPhotometricLight::Intensity 620
GonioPhotometricLight::lightPos 620
Light 606
Light::LightToWorld 606
MIPMap 530
Point 63
ProjectionLight 614
RGBSpectrum 279
Spectrum 263
Transform 76

⟨*GonioPhotometricLight Method Definitions*⟩ ≡

```
GonioPhotometricLight::GonioPhotometricLight(const Transform &light2world,
        const Spectrum &intensity, const string &texname)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    ⟨Create mipmap for GonioPhotometricLight 620⟩
}
```

Like `ProjectionLight`, `GonioPhotometricLight` constructs a `MIPMap` of the distribution's image map, also always using `RGBSpectrum` values.



Figure 12.9: Scene Rendered Using a Goniophotometric Diagram from Figure 12.8. Even though a point light source is the basis of this light, including the directional variation of a realistic light improves the visual realism of the rendered image.

⟨*Create* mipmap *for* GonioPhotometricLight⟩ ≡ 619
```
int width, height;
RGBSpectrum *texels = ReadImage(texname, &width, &height);
if (texels) {
    mipmap = new MIPMap<RGBSpectrum>(width, height, texels);
    delete[] texels;
}
else mipmap = NULL;
```

⟨*GonioPhotometricLight Private Data*⟩ ≡ 618
```
Point lightPos;
Spectrum Intensity;
MIPMap<RGBSpectrum> *mipmap;
```

The GonioPhotometricLight::Sample_L() method is not shown here. It is essentially identical to the SpotLight::Sample_L() and ProjectionLight::Sample_L() methods that use a helper function to scale the amount of radiance. It assumes that the scale texture is encoded using spherical coordinates, so that the given direction needs to be converted to θ and ϕ values and scaled to $[0, 1]$ before being used to index into the texture. Goniophotometric diagrams are usually defined in a coordinate space where the y axis is up, whereas the spherical coordinate utility routines assume that z is up, so y and z are swapped before doing the conversion.

GonioPhotometricLight::
mipmap 620
MIPMap 530
Point 63
ProjectionLight::
Sample_L() 617
ReadImage() 1004
RGBSpectrum 279
Spectrum 263
SpotLight::Sample_L() 713

⟨*GonioPhotometricLight Public Methods*⟩ ≡ 618
```
Spectrum Scale(const Vector &w) const {
    Vector wp = Normalize(WorldToLight(w));
    swap(wp.y, wp.z);
    float theta = SphericalTheta(wp);
    float phi   = SphericalPhi(wp);
    float s = phi * INV_TWOPI, t = theta * INV_PI;
    return (mipmap == NULL) ? 1.f :
           Spectrum(mipmap->Lookup(s, t, SPECTRUM_ILLUMINANT));
}
```

The `Power()` method's computation is inaccurate because the spherical coordinate parameterization of directions has various distortions, particularly near the $+z$ and $-z$ directions. Again, this error is acceptable for the uses of this method in `pbrt`.

⟨*GonioPhotometricLight Method Definitions*⟩ +≡
```
Spectrum GonioPhotometricLight::Power(const Scene *) const {
    return 4.f * M_PI * Intensity *
        Spectrum(mipmap->Lookup(.5f, .5f, .5f), SPECTRUM_ILLUMINANT);

}
```

12.3 DISTANT LIGHTS

Another useful light source type is the *distant light*, also known as a *directional light*. It describes an emitter that deposits illumination from the same direction at every point in space. Such a light is also called a point light "at infinity," since, as a point light becomes progressively farther away, it acts more and more like a directional light. For example, the sun (as considered from Earth) can be thought of as a directional light source. Although it is actually an area light source, the illumination effectively arrives at Earth in parallel beams because it is so far away. The `DistantLight`, implemented in the files `lights/distant.h` and `lights/distant.cpp`, implements a directional source.

DistantLight 621
DistantLight::L 622
DistantLight::lightDir 622
GonioPhotometricLight::Intensity 620
GonioPhotometricLight::mipmap 620
INV_PI 1002
INV_TWOPI 1002
Light 606
Light::LightToWorld 606
Light::WorldToLight 606
MIPMap::Lookup() 540
M_PI 1002
Scene 22
Spectrum 263
SPECTRUM_ILLUMINANT 277
SphericalPhi() 292
SphericalTheta() 292
Transform 76
Vector 57
Vector::Normalize() 63

⟨*DistantLight Declarations*⟩ ≡
```
class DistantLight : public Light {
public:
    ⟨DistantLight Public Methods⟩
private:
    ⟨DistantLight Private Data 622⟩
};
```

⟨*DistantLight Method Definitions*⟩ ≡
```
DistantLight::DistantLight(const Transform &light2world,
        const Spectrum &radiance, const Vector &dir)
    : Light(light2world) {
    lightDir = Normalize(LightToWorld(dir));
    L = radiance;
}
```

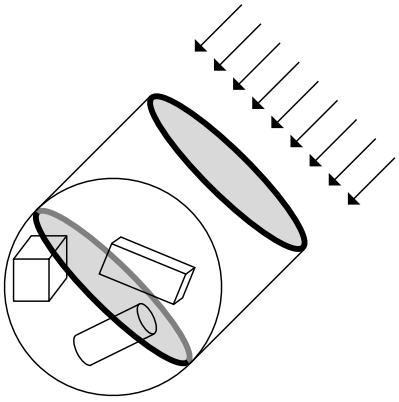


Figure 12.10: An approximation of the power emitted by a distant light into a given scene can be obtained by finding the sphere that bounds the scene, computing the area of an inscribed disk, and computing the power that arrives on the surface of that disk.

⟨*DistantLight Private Data*⟩ ≡ 621
```
Vector lightDir;
Spectrum L;
```

⟨*DistantLight Method Definitions*⟩ +≡
```
Spectrum DistantLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    *wi = lightDir;
    *pdf = 1.f;
    visibility->SetRay(p, pEpsilon, *wi, time);
    return L;
}
```

The distant light is unusual in that the amount of power it emits is related to the spatial extent of the scene. In fact, it is proportional to the area of the scene receiving light. To see why this is so, consider a disk of area A being illuminated by a distant light with emitted radiance L where the incident light arrives along the disk's normal direction. The total power reaching the disk is $\Phi = AL$. As the size of the receiving surface varies, power varies proportionally.

To find the emitted power for a `DistantLight`, it's impractical to compute the total surface area of the objects that are visible to the light. Instead, we will approximate this area with a disk inside the scene's bounding sphere oriented in the light's direction (Figure 12.10). This will always overestimate the actual area, but is sufficient for the needs of code elsewhere in the system.

DistantLight 621
DistantLight::L 622
DistantLight::lightDir 622
LightSample 710
Point 63
Spectrum 263
Vector 57
VisibilityTester 608
VisibilityTester::
SetRay() 609

⟨*DistantLight Method Definitions*⟩ +≡
```
Spectrum DistantLight::Power(const Scene *scene) const {
    Point worldCenter;
    float worldRadius;
    scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
    return L * M_PI * worldRadius * worldRadius;
}
```

12.4 AREA LIGHTS

Area lights are light sources defined by one or more Shapes that emit light from their surface, with some directional distribution of radiance at each point on the surface. In general, computing radiometric quantities related to area lights requires computing integrals over the surface of the light that often can't be computed in closed form. This issue is addressed with the Monte Carlo integration techniques of Chapter 14. The reward for this complexity (and computational expense) is soft shadows and more realistic lighting effects, rather than the hard shadows and stark lighting that come from point lights. (See Figure 12.11. Also compare Figure 12.12 to Figure 12.2.)

The AreaLight class is an abstract base class that inherits from Light. Implementations of area lights should inherit from it.

⟨*Light Declarations*⟩ +≡
```
class AreaLight : public Light {
public:
    ⟨AreaLight Interface 625⟩
};
```



Figure 12.11: Wider View of the Lighting Example Scene. The dragon is illuminated by a disk area light source directly above it.

AreaLight 623
BBox::BoundingSphere() 74
DistantLight::L 622
Light 606
M_PI 1002
Point 63
Scene 22
Scene::WorldBound() 24
Shape 108
Spectrum 263



(a)



(b)

Figure 12.12: Dragon Model Illuminated by Disk Area Lights. (a) The disk's radius is relatively small; the shadow has soft penumbrae, but otherwise the image looks similar to the one with a point light. (b) The effect of using a much larger disk: not only have the penumbrae become much larger, to the point of nearly eliminating the fully in-shadow areas, but notice how areas like the neck of the dragon and its jaw have noticeably different appearances when illuminated from a wider range of directions.

AreaLight adds a single new method to the Light interface, AreaLight::L(). Implementations are given a point on the surface of the light and the surface normal and should evaluate the area light's emitted radiance, L, in the given outgoing direction.

⟨*AreaLight Interface*⟩ ≡ 623
```
virtual Spectrum L(const Point &p, const Normal &n,
                   const Vector &w) const = 0;
```

For convenience, there is a method in the Intersection class that makes it easy to compute the emitted radiance at a surface point intersected by a ray. It gets the AreaLight pointer from the primitive and calls its L() method.

⟨*Intersection Method Definitions*⟩ +≡
```
Spectrum Intersection::Le(const Vector &w) const {
    const AreaLight *area = primitive->GetAreaLight();
    return area ? area->L(dg.p, dg.nn, w) : Spectrum(0.);
}
```

DiffuseAreaLight implements a basic area light source with a uniform spatial and directional radiance distribution. It only emits light on the side of the surface with outward-facing surface normal; there is no emission from the other side. (The Shape::ReverseOrientation value can be set to true to cause the light to be emitted from the other side of the surface instead.) DiffuseAreaLight is defined in the files lights/diffuse.h and lights/diffuse.cpp.

⟨*DiffuseAreaLight Declarations*⟩ ≡
```
class DiffuseAreaLight : public AreaLight {
public:
    ⟨DiffuseAreaLight Public Methods 626⟩
protected:
    ⟨DiffuseAreaLight Protected Data 626⟩
};
```

AreaLight 623
AreaLight::L() 625
DifferentialGeometry::p 102
DiffuseAreaLight 625
Intersection 186
Intersection::dg 186
Light 606
Normal 65
Point 63
Primitive::GetAreaLight() 187
Shape 108
Shape::CanIntersect() 110
Shape::ReverseOrientation 108
ShapeSet 626
Spectrum 263
Vector 57

The constructor caches the surface area of the light source since this area calculation may be computationally expensive. However, not all shapes can compute their surface area (one example is subdivision surfaces). More generally, some shapes can't be used as area lights directly since they don't implement all of the Shape methods that the AreaLight needs to call. For example, in the Monte Carlo sampling code in Chapter 14, DiffuseAreaLight methods implemented there will need to call Shape methods that sample points on the shape's surface; these methods cannot be implemented for some types of shapes.

Such shapes must be refined into simpler shapes until their CanIntersect() methods return true.[2] This refinement is done in the DiffuseAreaLight constructor if needed. The eventual shape or shapes that are created are stored in a container class called ShapeSet. This class implements the Shape interface, so that the DiffuseAreaLight can be written to store only a single Shape, thus simplifying its implementation.

2 We are thus overloading the semantics of the Shape::CanIntersect() method to include both the ability to compute intersections, the ability to compute surface area, and the ability to generate samples.

⟨*DiffuseAreaLight Method Definitions*⟩ ≡
```
DiffuseAreaLight::DiffuseAreaLight(const Transform &light2world,
        const Spectrum &le, int ns, const Reference<Shape> &s)
    : AreaLight(light2world, ns) {
    Lemit = le;
    shapeSet = new ShapeSet(s);
    area = shapeSet->Area();
}
```

⟨*DiffuseAreaLight Protected Data*⟩ ≡ 625
```
Spectrum Lemit;
ShapeSet *shapeSet;
float area;
```

Because this area light implementation emits light from only one side of the shape's surface, its `L()` method just makes sure that the outgoing direction lies in the same hemisphere as the normal.

⟨*DiffuseAreaLight Public Methods*⟩ ≡ 625
```
Spectrum L(const Point &p, const Normal &n, const Vector &w) const {
    return Dot(n, w) > 0.f ? Lemit : 0.f;
}
```

The `ShapeSet` class is a simple wrapper for a vector of shapes. Most of its implementation is in Section 14.6.4, where its methods related to random sampling collections of shapes for Monte Carlo integration are implemented.

⟨*ShapeSet Declarations*⟩ ≡
```
class ShapeSet {
public:
    ⟨ShapeSet Public Methods⟩
private:
    ⟨ShapeSet Private Data 626⟩
};
```

⟨*ShapeSet Private Data*⟩ ≡ 626
```
vector<Reference<Shape> > shapes;
```

The `DiffuseAreaLight::Sample_L()` method isn't as straightforward as it has been for the other light sources described so far. Specifically, at each point in the scene, radiance from area lights can be incident from many directions, not just a single direction as was the case for the other lights (Figure 12.13). This leads to the question of which direction should be chosen for this method. We will defer answering this question and providing an implementation of this method until Section 14.6, after Monte Carlo integration has been introduced.

Emitted power from an area light with uniform emitted radiance over the surface can be directly computed in closed form:

AreaLight 623
DiffuseAreaLight 625
DiffuseAreaLight::area 626
DiffuseAreaLight::Lemit 626
DiffuseAreaLight::Sample_L() 718
Dot() 60
Normal 65
Point 63
Reference 1011
Shape 108
ShapeSet 626
Spectrum 263
Transform 76
Vector 57

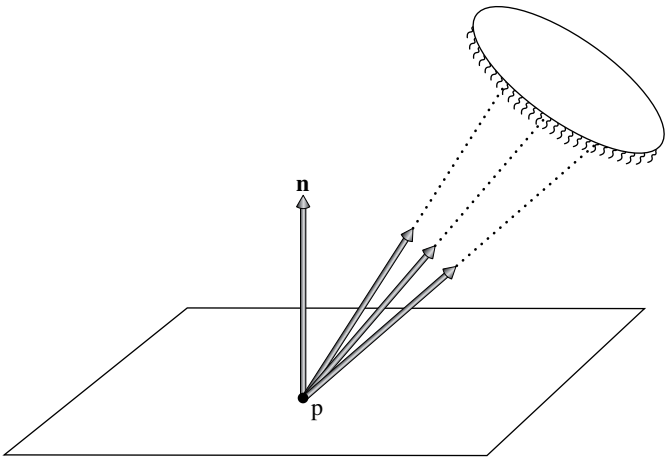


Figure 12.13: An area light casts incident illumination along many directions, rather than from a single direction.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
Spectrum DiffuseAreaLight::Power(const Scene *) const {
    return Lemit * area * M_PI;
}
```

Finally, area lights clearly do not represent singularities, so they return `false` from `IsDeltaLight()`:

⟨*DiffuseAreaLight Public Methods*⟩ +≡ 625
```
bool IsDeltaLight() const { return false; }
```

12.5 INFINITE AREA LIGHTS

Another useful kind of light is the infinite area light—an infinitely faraway area light source that surrounds the entire scene. One way to visualize this light is as an enormous sphere that casts light into the scene from every direction. One important use of infinite area lights is for *environment lighting*, where an image that represents illumination in an environment is used to illuminate synthetic objects as if they were in that environment. Figures 12.14 and 12.15 compare illuminating a car model with a standard area light to illuminating it with environment maps that simulate illumination from the sky at a few different times of day (the illumination maps used are shown in Figure 12.16). The increase in realism is striking. The `InfiniteAreaLight` class is implemented in `lights/infinite.h` and `lights/infinite.cpp`.

A widely used representation for light for this application is the latitude–longitude radiance map. The `EnvironmentCamera` can be used to create image maps for the light, or see the "Further Reading" section for information about techniques for capturing this

DiffuseAreaLight::area 626
DiffuseAreaLight::Lemit 626
EnvironmentCamera 318
InfiniteAreaLight 629
M_PI 1002
Scene 22
Spectrum 263



(a)



(b)

Figure 12.14: Car model (a) illuminated with an area light and a directional light and (b) illuminated with morning skylight from an environment map. Using a realistic distribution of illumination gives an image that is much more visually compelling. In particular, with illumination arriving from all directions, the glossy reflective properties of the paint are much more visually apparent; in (a) there isn't anything to reflect, so the paint inaccurately appears to be dull matte.

lighting data from real-world environments. Like the other lights, the `InfiniteAreaLight` takes a transformation matrix; here, its use is to orient the image map. It then uses spherical coordinates to map from directions on the sphere to (θ, ϕ) directions, and from there to (u, v) texture coordinates. The provided transformation thus determines which direction is "up."

InfiniteAreaLight 629



(a)



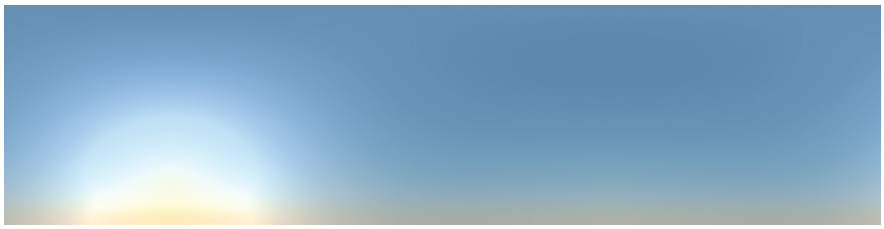
(b)

Figure 12.15: Changing just the environment map used for illumination gives quite different results in the final image: (a) using a midday skylight distribution and (b) using a sunset environment map.

⟨*InfiniteAreaLight Declarations*⟩ ≡

```
class InfiniteAreaLight : public Light {
public:
    ⟨InfiniteAreaLight Public Methods 631⟩
private:
    ⟨InfiniteAreaLight Private Data 631⟩
};
```

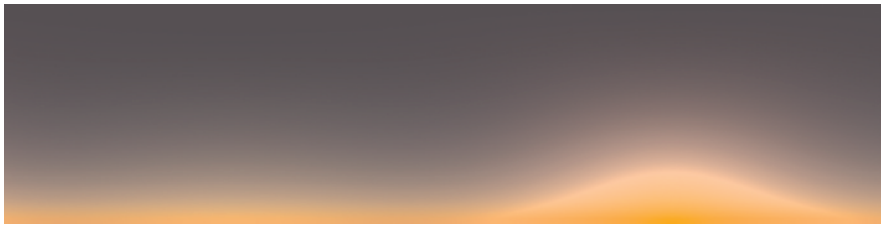
Light 606



(a)



(b)



(c)

Figure 12.16: Environment Maps Used for Illumination in Figures 12.14 and 12.15. (a) Morning, (b) midday, and (c) sunset sky. (The bottom halves of these maps aren't shown here, since they are just black pixels.)

The constructor loads the image data from disk and creates a `MIPMap` to store it. The fragment that loads the data, ⟨*Read texel data from* `texmap` *into* `texels`⟩, is straightforward and won't be included here. The other code fragment in the constructor, ⟨*Initialize sampling PDFs for infinite area light*⟩, is related to Monte Carlo sampling of `InfiniteAreaLights` and will be defined later, in Section 14.6.5.

⟨*InfiniteAreaLight Method Definitions*⟩ ≡

```
InfiniteAreaLight::InfiniteAreaLight(const Transform &light2world,
        const Spectrum &L, int ns, const string &texmap)
    : Light(light2world, ns) {
    int width = 0, height = 0;
    RGBSpectrum *texels = NULL;
    ⟨Read texel data from texmap into texels⟩
```

InfiniteAreaLight 629
InfiniteAreaLight::
radianceMap 631
Light 606
MIPMap 530
RGBSpectrum 279
Spectrum 263
Transform 76

```
    radianceMap = new MIPMap<RGBSpectrum>(width, height, texels);
    delete[] texels;
    ⟨Initialize sampling PDFs for infinite area light 725⟩
}
```

⟨*InfiniteAreaLight Private Data*⟩ ≡ 629
```
MIPMap<RGBSpectrum> *radianceMap;
```

Because `InfiniteAreaLights` cast light from all directions, it's also necessary to use Monte Carlo integration to compute reflected light due to illumination from them. Therefore, the `InfiniteAreaLight::Sample_L()` method will be defined in Section 14.6.

Like directional lights, the total power from the infinite area light is related to the surface area of the scene:

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Power(const Scene *scene) const {
    Point worldCenter;
    float worldRadius;
    scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
    return M_PI * worldRadius * worldRadius *
        Spectrum(radianceMap->Lookup(.5f, .5f, .5f), SPECTRUM_ILLUMINANT);
}
```

BBox::BoundingSphere() 74
InfiniteAreaLight 629
InfiniteAreaLight::radianceMap 631
InfiniteAreaLight::Sample_L() 727
INV_PI 1002
INV_TWOPI 1002
Light 606
Light::WorldToLight 606
MIPMap 530
MIPMap::Lookup() 540
M_PI 1002
Point 63
Ray::d 67
RayDifferential 69
RGBSpectrum 279
Scene 22
Scene::WorldBound() 24
Spectrum 263
SPECTRUM_ILLUMINANT 277
SphericalPhi() 292
SphericalTheta() 292
Vector 57
Vector::Normalize() 63

⟨*InfiniteAreaLight Public Methods*⟩ ≡ 629
```
bool IsDeltaLight() const { return false; }
```

Because infinite area lights need to be able to contribute radiance to rays that don't hit any geometry in the scene, we'll add a method to the base `Light` class that returns emitted radiance due to that light along a ray that didn't hit anything in the scene. (The default implementation for other lights returns no radiance.) It is the responsibility of the integrators to call this method for these rays.

⟨*Light Method Definitions*⟩ +≡
```
Spectrum Light::Le(const RayDifferential &) const {
    return Spectrum(0.);
}
```

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Le(const RayDifferential &r) const {
    Vector wh = Normalize(WorldToLight(r.d));
    float s = SphericalPhi(wh) * INV_TWOPI;
    float t = SphericalTheta(wh) * INV_PI;
    return Spectrum(radianceMap->Lookup(s, t), SPECTRUM_ILLUMINANT);
}
```

FURTHER READING

Warn (1983) developed early models of light sources with nonisotropic emission distributions, including the spotlight model used in this chapter. Verbeck and Greenberg (1984) also described a number of techniques for modeling light sources that are now classic parts of the light modeling toolbox. More recently, Barzel (1997) described a highly parameterized model for light sources, including many controls for controlling rate of falloff, the area of space that is illuminated, and so on. Bjorke (2001) described a number of additional controls for controlling illumination for artistic effect. (The Barzel and Bjorke approaches are not physically based, which isn't a problem in most applications outside of physically based rendering.)

Blinn and Newell (1976) first introduced the idea of environment maps and their use for simulating illumination, although they only considered illumination of specular objects. Greene (1986) further refined these ideas, considering antialiasing and different representations for environment maps. Nishita and Nakamae (1986) developed algorithms for efficiently rendering objects illuminated by hemispherical skylights and generated some of the first images that showed off that distinctive lighting effect.

Miller and Hoffman (1984) were the first to consider using arbitrary environment maps to illuminate objects with diffuse and glossy BRDFs. Debevec (1998) later extended this work and investigated issues related to capturing images of real environments. The *HDRShop* tool, available from *www.debevec.org/HDRShop*, is invaluable for converting among different environment map formats and parameterizations.

Algorithms to render soft shadows from area lights were first developed by Amanatides (1984) and Cook, Porter, and Carpenter (1984). See both Hasenfratz et al. (2003) and Akenine-Möller et al. (2008) for overviews of more recent algorithms for rendering soft shadows, particularly for interactive rendering.

The goniometric light source approximation is widely used to model area light sources in the field of illumination engineering. The rule of thumb there is that once a point is five times an area light source's radius away from it, a point light approximation has sufficient accuracy for most applications. Ashdown (1993) discusses this application in depth and proposes a more sophisticated light source representation for accurate illumination computations. A variety of file format standards have been developed for encoding goniophotometric diagrams for these applications (Illuminating Engineering Society of North America Computer Committee 1986; Stockmar 1986). Many lighting fixture manufacturers provide data in these formats on their Web sites. Another generalization of goniometric lights was suggested by Heidrich et al. (1998), who represented light sources as a 4D exitant *lightfield*—essentially a function of both position and direction.

As discussed previously, one way to reduce the time spent tracing shadow rays is to have methods like `Shape::IntersectP()` and `Primitive::IntersectP()` that just check for any occlusion along a ray without bothering to compute the geometric information at the intersection point. Other approaches for optimizing ray tracing for shadow rays include the *shadow cache*, where each light stores a pointer to the last primitive that occluded a shadow ray to the light where that primitive is checked first to see if it occludes subsequent shadow rays before the ray is passed to the acceleration structure (Haines and

`Primitive::IntersectP()` 186
`Shape::IntersectP()` 111

Greenberg 1986). Pearce (1991) pointed out that the shadow cache doesn't work well if the scene has finely tessellated geometry; it may be better to cache the voxel of BVH node that held the last occluder, for instance. (The shadow cache can similarly be defeated when multiple levels of reflection and refraction are present or when Monte Carlo ray-tracing techniques are used.) Hart, Dutré, and Greenberg (1999) developed a generalization of the shadow cache, which tracks which objects block light from particular light sources and clips their geometry against the light source geometry so that shadow rays don't need to be traced toward the parts of the light that are certain to be occluded.

A related technique, described by Haines and Greenberg (1986), is the light buffer for point light sources, where the light discretizes the directions around it and determines which objects are visible along each set of directions (and are thus potential occluding objects for shadow rays). Another effective optimization is *shaft culling*, which takes advantage of coherence among groups of rays traced in a similar set of directions (e.g., shadow rays from a single point to points on an area light source). With shaft culling, a shaft that bounds a collection of rays is computed and then the objects in the scene that penetrate the shaft are found. For all of the rays in the shaft, it is only necessary to check for intersections with those objects that intersect the shaft, and the expense of ray intersection acceleration structure traversal for each of the rays is avoided (Haines and Wallace 1994).

Woo and Amanatides (1990) classified which lights are visible, not visible, and partially visible in different parts of the scene and stored this information in a voxel-based 3D data structure, using this information to save shadow ray tests. Fernandez, Bala, and Greenberg (2002) developed a similar approach based on spatial decomposition that stores references to important blockers in each voxel and also builds up this information on demand for applications like walkthroughs.

For complex models, simplified versions of their geometry can be used for shadow ray intersections. For example, the simplification envelopes described by Cohen et al. (1996) can create a simplified mesh that bounds a given mesh from both the inside and the outside. If a ray misses the mesh that bounds a complex model from the outside, or intersects the mesh that bounds it from the inside, then no further shadow processing is necessary. Only the uncertain remaining cases need to be intersected against the full geometry. A related technique is described by Lukaszewski (2001), who uses the Minkowski sum to effectively expand primitives (or bounds of primitives) in the scene so that intersecting one ray against one of these primitives can determine if any of a collection of rays might have intersected the actual primitives.

EXERCISES

● 12.1 Shadow mapping is a technique for rendering shadows from point and distant light sources based on rendering an image from the light source's perspective that records depth in each pixel of the image and then projecting points onto the shadow map and comparing their depth to the depth of the first visible object as seen from the light in that direction. This method was first described by Williams (1978), and Reeves, Salesin, and Cook (1987) developed a number of key improvements. Modify `pbrt` to be able to render depth map images into

a file and then use them for shadow testing for lights in place of tracing shadow rays. How much faster can this be? Discuss the advantages and disadvantages of the two approaches.

❶ 12.2 Through algebraic manipulation and precomputation of one more value in the constructor, the `SpotLight::Falloff()` method can be rewritten to compute the exact same result (modulo floating-point differences) while using no square root computations and no divides (recall that the `Vector::Normalize()` method performs both a square root and a divide). Derive and implement this optimization. How much is running time improved on a spotlight-heavy scene?

❸ 12.3 The current light source implementations don't support animated transformations. Modify `pbrt` to include this functionality and render images showing off the effect of animating light positions.

❷ 12.4 Modify the `ProjectionLight` to also support orthographic projections. This variant is particularly useful even without an image map, since it gives a directional light source with a beam of user-defined extent.

❷ 12.5 Write an `AreaLight` implementation that improves on the `DiffuseAreaLight` by supporting spatially and directionally varying emitted radiance, specified via either image maps or `Textures`. Use it to render images with effects like a television illuminating a dark room or a stained-glass window lit from behind.

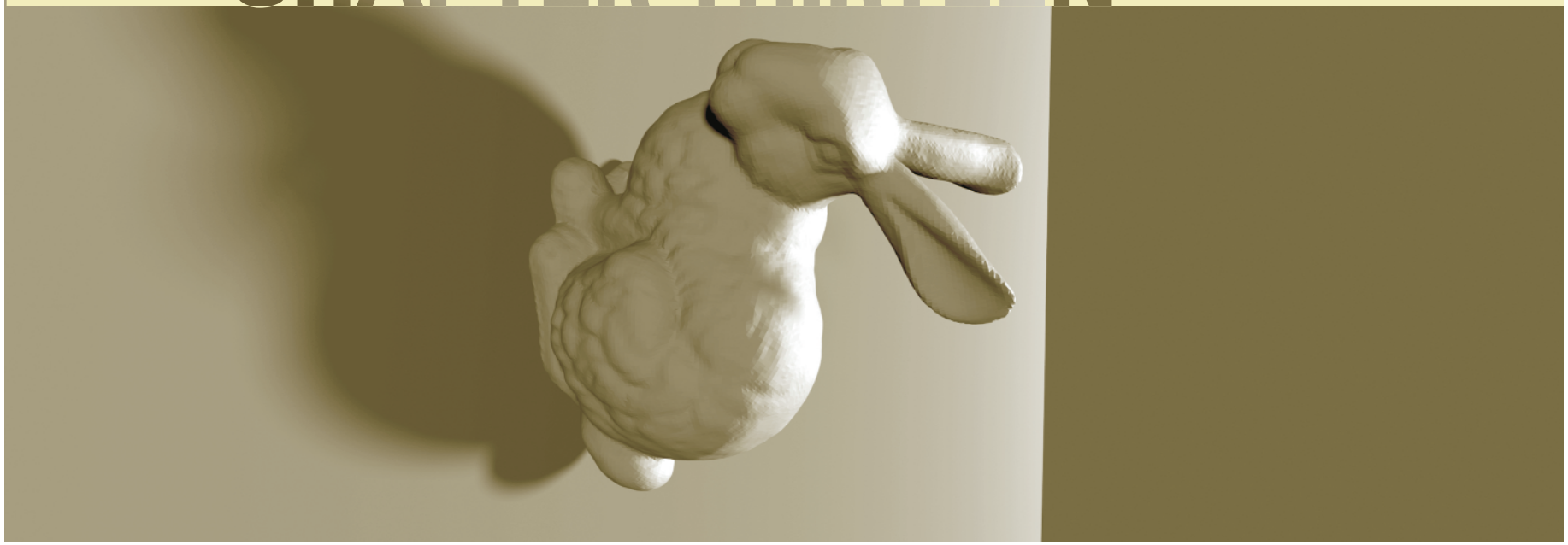
❷ 12.6 Read some of the papers in the "Further Reading" section that discuss the shadow cache and add this optimization to `pbrt`. Measure how much it speeds up the system for a variety of scenes. What techniques can you come up with that make it work better in the presence of multiple levels of reflection?

❸ 12.7 Modify `pbrt` to support the shaft culling algorithm (Haines and Wallace 1994). Measure the performance difference for scenes with area light sources. Make sure that your implementation still performs well even with very large light sources (like a hemispherical skylight).

AreaLight 623
DiffuseAreaLight 625
ProjectionLight 614
SpotLight::Falloff() 614
Texture 519
Vector::Normalize() 63

This page intentionally left blank

CHAPTER THIRTEEN



13 MONTE CARLO INTEGRATION I: BASIC CONCEPTS

Before we introduce the `SurfaceIntegrators` and `VolumeIntegrators` that compute radiance along rays arriving at the camera, we will first lay some groundwork regarding the techniques they will use to compute solutions to the integral equations that describe light scattering. These integral equations generally do not have analytic solutions, so we must turn to numerical methods. Although standard numerical integration techniques like trapezoidal integration or Gaussian quadrature are very effective at solving low-dimensional smooth integrals, their rate of convergence for the higher-dimensional and discontinuous integrals that are common in rendering is poor.

Monte Carlo numerical integration methods provide one solution to this problem. They use randomness to evaluate integrals with a convergence rate that is independent of the dimensionality of the integrand. In this chapter, we review important concepts from probability and lay the foundation for using Monte Carlo techniques to evaluate the key integrals in rendering. The following chapter will then describe techniques for improving the convergence rate of these approaches and will describe the specific algorithms used in `pbrt`.

Judicious use of randomness has revolutionized the field of algorithm design. Randomized algorithms fall broadly into two classes: *Las Vegas* and *Monte Carlo*. Las Vegas algorithms are those that use randomness but always give the same result in the end (e.g., choosing a random array entry as the pivot element in Quicksort). Monte Carlo algorithms, on the other hand, give different results depending on the particular random numbers used along the way, but give the right answer *on average*. So, by averaging the results of several runs of a Monte Carlo algorithm (on the same input), it is possible to find a result that is statistically very likely to be close to the true answer. Motwani and

Raghavan (1995) have written an excellent introduction to the field of randomized algorithms.

Monte Carlo integration[1] is a method for using random sampling to estimate the values of integrals. One very useful property of Monte Carlo is that one needs only to be able to evaluate the integrand at arbitrary points in the domain in order to estimate the value of its integral $\int f(x)\,\mathrm{d}x$. This property not only makes Monte Carlo easy to implement but also makes the technique applicable to a broad variety of integrands, including those containing discontinuities.

Many of the integrals that arise in rendering are difficult or impossible to evaluate directly. For example, to compute the amount of light reflected by a surface at a point, Equation (5.8), we must integrate the product of the incident radiance and the BSDF over the unit sphere. Because object visibility and thus the incident radiance function at a point in a complex scene varies in difficult-to-predict ways, a closed-form expression for all of the terms in this product is almost never available, and even if it were, performing the integral analytically is generally not possible. Monte Carlo integration makes it possible to estimate the reflected radiance simply by choosing a set of directions over the sphere, computing the incident radiance along them, multiplying by the BSDF's value for those directions, and applying a weighting term. Arbitrary BSDFs, light source descriptions, and scene geometry are easily handled; evaluation of each of these functions at arbitrary points is all that is required.

The main disadvantage of Monte Carlo is that if n samples are used to estimate the integral, the algorithm converges to the correct result at a rate of $O(n^{-1/2})$. In other words, to cut the error in half, it is necessary to evaluate four times as many samples. In rendering, each sample generally requires that one or more rays be traced in the process of computing the value of the integrand, a computationally-expensive cost to bear when using Monte Carlo for image synthesis. In images, artifacts from Monte Carlo sampling manifest themselves as noise—pixels are randomly too bright or too dark. Most of the current research in Monte Carlo for computer graphics is about reducing this error as much as possible while minimizing the number of additional samples that must be taken.

13.1 BACKGROUND AND PROBABILITY REVIEW

We will start by defining some terms and reviewing basic ideas from probability. We assume that the reader is already familiar with basic probability concepts; readers needing a more complete introduction to this topic should consult a textbook such as Sheldon Ross's *Introduction to Probability Models* (2002).

A *random variable* X is a value chosen by some random process. We will generally use capital letters to denote random variables, with exceptions made for a few Greek symbols that represent special random variables. Random variables are always drawn from some domain, which can be either discrete (e.g., a fixed set of possibilities) or continuous (e.g.,

1 For brevity, we will refer to Monte Carlo integration simply as "Monte Carlo."

the real numbers $\mathbb{R}$). Applying a function f to a random variable X results in a new random variable $Y = f(X)$.

For example, the result of a roll of a die is a discrete random variable sampled from the set of events $X_i = \{1, 2, 3, 4, 5, 6\}$. Each event has a probability $p_i = \frac{1}{6}$, and the sum of probabilities $\sum p_i$ is necessarily one. We can take a continuous, uniformly distributed random variable $\xi \in [0, 1)$ and map it to a discrete random variable, choosing X_i if

$$\sum_{j=1}^{i-1} p_j < \xi \leq \sum_{j=1}^{i} p_j.$$

For lighting applications, we might want to define the probability of sampling illumination from each light in the scene based on the power Φ_i from each source relative to the total power from all sources:

$$p_i = \frac{\Phi_i}{\sum_j \Phi_j}.$$

Notice that these p_i also sum to one.

The *cumulative distribution function* (CDF) $P(x)$ of a random variable is the probability that a value from the variable's distribution is less than or equal to some value x:

$$P(x) = Pr\{X \leq x\}.$$

For the die example, $P(2) = \frac{1}{3}$, since two of the six possibilities are less than or equal to 2.

13.1.1 CONTINUOUS RANDOM VARIABLES

In rendering, discrete random variables are less common than continuous random variables, which take on values over ranges of continuous domains (e.g., the real numbers or directions on the unit sphere).

A particularly important random variable is the *canonical uniform random variable*, which we will write as ξ. This variable takes on all values in its domain $[0, 1)$ with equal probability. This particular variable is important for two reasons. First, it is easy to generate a variable with this distribution in software—most run time libraries have a pseudo-random number generator that does just that.[2] Second, as we will show later, it is possible to generate samples from arbitrary distributions by first starting with canonical uniform random variables and applying an appropriate transformation. The technique described previously for mapping from ξ to the six faces of a die gives a flavor of this technique in the discrete case.

Another example of a continuous random variable is one that ranges over the real numbers between 0 and 2, where the probability of it taking on any particular value x is proportional to the value $2 - x$: it is twice as likely for this random variable to take on

2 Although the theory of Monte Carlo is based on using truly random numbers, in practice a well-written pseudo-random number generator (PRNG) is sufficient. `pbrt` uses a particularly high-quality PRNG that returns a sequence of pseudo-random values that is effectively as "random" as true random numbers. (Many PRNGs are not as well implemented and have detectable patterns in the sequence of numbers they generate.) True random numbers, found by measuring random phenomena like atomic decay or atmospheric noise, are available from sources like *www.random.org* for those for whom PRNGs are not acceptable.

a value around zero as it is to take one around one, and so forth. The *probability density function* (PDF) formalizes this idea: it describes the relative probability of a random variable taking on a particular value. The PDF $p(x)$ is the derivative of the random variable's CDF,

$$p(x) = \frac{\mathrm{d}P(x)}{\mathrm{d}x}.$$

For uniform random variables, $p(x)$ is a constant; this is a direct consequence of uniformity. For ξ we have

$$p(x) = \begin{cases} 1 & x \in [0, 1) \\ 0 & \text{otherwise.} \end{cases}$$

PDFs are necessarily nonnegative and always integrate to one over their domains. Given an arbitrary interval $[a, b]$ in the domain, the PDF can give the probability that a random variable lies inside the interval:

$$P(x \in [a, b]) = \int_a^b p(x)\, \mathrm{d}x.$$

This follows directly from the first fundamental theorem of calculus and the definition of the PDF.

13.1.2 EXPECTED VALUES AND VARIANCE

The *expected value* $E_p[f(x)]$ of a function f is defined as the average value of the function over some distribution of values $p(x)$ over its domain. In the next section, we will see how Monte Carlo integration computes the expected values of arbitrary integrals. Expected value over a domain, D, is defined as

$$E_p[f(x)] = \int_D f(x)\, p(x)\, \mathrm{d}x. \tag{13.1}$$

As an example, consider the problem of finding the expected value of the cosine function between 0 and π, where p is uniform.[3] Because the PDF $p(x)$ must integrate to one over the domain, $p(x) = 1/\pi$, so

$$E[\cos x] = \int_0^\pi \frac{\cos x}{\pi}\, \mathrm{d}x = \frac{1}{\pi}(-\sin \pi + \sin 0) = 0,$$

which is precisely the expected result. (Consider the graph of $\cos x$ over $[0, \pi]$ to see why this is so.)

The *variance* of a function is the expected deviation of the function from its expected value. Variance is a fundamental concept for quantifying the error in a value estimated by a Monte Carlo algorithm. It provides a precise way to quantify this error and measure how improvements to Monte Carlo algorithms reduce the error in the final result. Most of Chapter 14 is devoted to techniques for reducing variance and thus improving the

3 When computing expected values with a uniform distribution, we will drop the subscript p from E_p.

results computed by pbrt. The variance of a function f is defined as

$$V[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right].$$

The expected value and variance have three important properties that follow immediately from their respective definitions:

$$E[af(x)] = aE[f(x)]$$
$$E\left[\sum_i f(X_i)\right] = \sum_i E[f(X_i)]$$
$$V[af(x)] = a^2 V[f(x)].$$

These properties, and some simple algebraic manipulation, yield a much simpler expression for the variance:

$$V[f(x)] = E\left[(f(x))^2\right] - E[f(x)]^2. \tag{13.2}$$

Thus, the variance is the expected value of the square minus the square of the expected value. Given random variables that are *independent*, variance also has the property that the sum of the variances is equal to the variance of their sum:

$$\sum_i V[f(X_i)] = V\left[\sum_i f(X_i)\right].$$

13.2 THE MONTE CARLO ESTIMATOR

We can now define the basic Monte Carlo estimator, which approximates the value of an arbitrary integral. It is the foundation of the light transport algorithms defined in Chapters 15 to 17.

Suppose that we want to evaluate a one-dimensional integral $\int_a^b f(x)\,\mathrm{d}x$. Given a supply of uniform random variables $X_i \in [a, b]$, the Monte Carlo estimator says that the expected value of the estimator

$$F_N = \frac{b-a}{N}\sum_{i=1}^{N} f(X_i),$$

$E[F_N]$, is in fact equal to the integral.[4] This fact can be demonstrated with just a few steps. First, note that the PDF $p(x)$ corresponding to the random variable X_i must be equal to $1/(b-a)$, since p must both be a constant and also integrate to one over the domain $[a, b]$. Algebraic manipulation then shows that

Lerp() 1000

RNG::RandomFloat() 1003

4 For example, the samples X_i might be computed in an implementation by Lerp(rng.RandomFloat(), a, b).

$$
\begin{aligned}
E[F_N] &= E\left[\frac{b-a}{N}\sum_{i=1}^{N} f(X_i)\right] \\
&= \frac{b-a}{N}\sum_{i=1}^{N} E\left[f(X_i)\right] \\
&= \frac{b-a}{N}\sum_{i=1}^{N}\int_a^b f(x)\ p(x)\ \mathrm{d}x \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)\ \mathrm{d}x \\
&= \int_a^b f(x)\ \mathrm{d}x.
\end{aligned}
$$

The restriction to uniform random variables can be relaxed with a small generalization. This is an extremely important step, since carefully choosing the PDF from which samples are drawn is an important technique for reducing variance in Monte Carlo (Section 14.4). If the random variables X_i are drawn from some arbitrary PDF $p(x)$, then the estimator

$$
F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{13.3}
$$

can be used to estimate the integral instead. The only limitation on $p(x)$ is that it must be nonzero for all x where $|f(x) > 0|$. It is similarly easy to see that the expected value of this estimator is the desired integral of f:

$$
\begin{aligned}
E[F_N] &= E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b \frac{f(x)}{p(x)}p(x)\ \mathrm{d}x \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)\ \mathrm{d}x \\
&= \int_a^b f(x)\ \mathrm{d}x.
\end{aligned}
$$

Extending this estimator to multiple dimensions or complex integration domains is straightforward. N samples X_i are taken from a multidimensional (or "joint") PDF, and the estimator is applied as usual. For example, consider the three-dimensional integral

$$
\int_{x_0}^{x_1}\int_{y_0}^{y_1}\int_{z_0}^{z_1} f(x, y, z)\ \mathrm{d}x\ \mathrm{d}y\ \mathrm{d}z.
$$

If samples $X_i = (x_i, y_i, z_i)$ are chosen uniformly from the box from (x_0, y_0, z_0) to (x_1, y_1, z_1), the PDF $p(X)$ is the constant value

$$\frac{1}{(x_1 - x_0)} \frac{1}{(y_1 - y_0)} \frac{1}{(z_1 - z_0)},$$

and the estimator is

$$\frac{(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)}{N} \sum_i f(X_i).$$

Note that the number of samples N can be chosen arbitrarily, regardless of the dimension of the integrand. This is another important advantage of Monte Carlo over traditional deterministic quadrature techniques. The number of samples taken in Monte Carlo is completely independent of the dimensionality of the integral, while with standard numerical quadrature techniques the number of samples required is exponential in the dimension.

Showing that the Monte Carlo estimator converges to the right answer is not enough to justify its use; a good rate of convergence is important too. Although we will not derive its rate of convergence here, it has been shown that error in the Monte Carlo estimator decreases at a rate of $O(\sqrt{N})$ in the number of samples taken. An accessible treatment of this topic can be found in Veach's thesis (Veach 1997, p. 39). Although standard quadrature techniques converge faster than $O(\sqrt{N})$ in one dimension, their performance becomes exponentially worse as the dimensionality of the integrand increases, while Monte Carlo's convergence rate is independent of the dimension, making Monte Carlo the only practical numerical integration algorithm for high-dimensional integrals. We have already encountered some high-dimensional integrals in this book, and in Chapter 15 we will see that the path tracing formulation of the light transport equation is an *infinite-dimensional* integral!

13.3 BASIC SAMPLING OF RANDOM VARIABLES

In order to evaluate the Monte Carlo estimator in Equation (13.3), it is necessary to be able to draw random samples from the chosen probability distribution. This section will introduce the basics of this process and demonstrate it with some straightforward examples. The next two sections will introduce more complex approaches to sampling before Section 13.6 develops the approach for the general multidimensional case. Sections 14.5 and 14.6 in the next chapter will show how to use these techniques to generate samples from the distributions defined by BSDFs and light sources.

13.3.1 THE INVERSION METHOD

The inversion method uses one or more uniform random variables and maps them to random variables from the desired distribution. To explain how this process works in general, we will start with a simple discrete example. Suppose we have a process with four possible outcomes. The probabilities of each of the four outcomes are given by p_1, p_2, p_3, and p_4, respectively, with the requirement that $\sum_{i=1}^{4} p_i = 1$. The corresponding PDF is shown in Figure 13.1.

In order to draw a sample from this distribution, we first find the CDF $P(x)$. In the continuous case, P is the indefinite integral of p. In the discrete case, we can directly construct the CDF by stacking the bars on top of each other, starting at the left. This idea

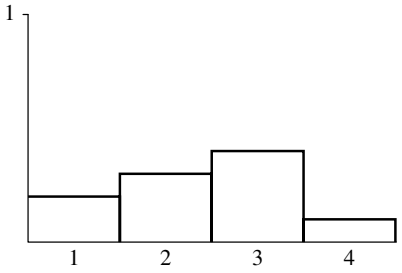


Figure 13.1: A Discrete PDF for Four Events Each with a Probability p_i. The sum of their probabilities $\sum_i p_i$ is necessarily one.

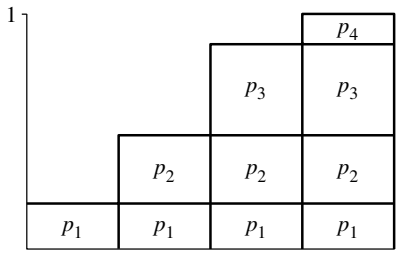


Figure 13.2: A Discrete CDF, Corresponding to the PDF in Figure 13.1. Each column's height is given by the PDF for the event that it represents plus the sum of the PDFs for the previous events, $P_i = \sum_{j=1}^{i} p_i$.

is shown in Figure 13.2. Notice that the height of the rightmost bar must be one because of the requirement that all probabilities sum to one.

To draw a sample from the distribution, we then take a uniform random number ξ and use it to select one of the possible outcomes using the CDF, doing so in a way that chooses a particular outcome with probability equal to its own probability. This idea is illustrated in Figure 13.3, where the events' probabilities are projected onto the vertical axis and a random variable ξ selects among them. It should be clear that this draws from the correct distribution—the probability of the uniform sample hitting any particular bar is exactly equal to the height of that bar. In order to generalize this technique to continuous distributions, consider what happens as the number of discrete possibilities approaches infinity. The PDF from Figure 13.1 becomes a smooth curve, and the CDF from Figure 13.2 becomes its integral. The projection process is still the same, although if the function is continuous, the projection has a convenient mathematical interpretation—it represents inverting the CDF and evaluating the inverse at ξ. This technique is thus called the *inversion method*.

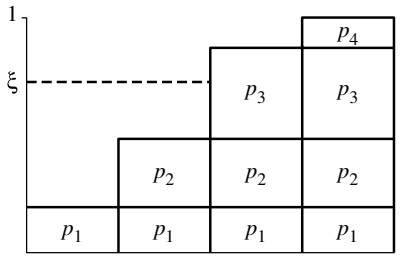


Figure 13.3: To use the inversion method to draw a sample from the distribution described by the PDF in Figure 13.1, a canonical uniform random variable is plotted on the vertical axis. By construction, the horizontal extension of ξ will intersect the box representing the ith outcome with probability p_i. If the corresponding event is chosen for a set of random variables ξ, then the resulting distribution of events will be distributed according to the PDF.

More precisely, we can draw a sample X_i from an arbitrary PDF $p(x)$ with the following steps:

1. Compute the CDF[5] $P(x) = \int_0^x p(x')\,\mathrm{d}x'$.
2. Compute the inverse $P^{-1}(x)$.
3. Obtain a uniformly distributed random number ξ.
4. Compute $X_i = P^{-1}(\xi)$.

Example: Power Distribution

As an example of how this procedure works, consider the task of drawing samples from a *power distribution*, $p(x) \propto x^n$. Sampling from this distribution will be performed when we are trying to sample the Blinn microfacet model. The PDF of the power distribution is

$$p(x) = cx^n,$$

for some constant c. The first task to tackle is to find the PDF for the function. In most cases, this simply involves computing the value of the proportionality constant c, which can be found using the constraint that $\int p(x)\,\mathrm{d}x = 1$:

$$\begin{aligned}
\int_0^1 cx^n\,\mathrm{d}x &= 1 \\
c\,\frac{x^{n+1}}{n+1}\bigg|_0^1 &= 1 \\
\frac{c}{n+1} &= 1 \\
c &= n+1.
\end{aligned}$$

5 In general, the lower limit of integration should be $-\infty$, although if $p(x) = 0$ for $x < 0$, this equation is equivalent.

Therefore, $p(x) = (n+1)x^n$. We can integrate this to get the CDF:

$$P(x) = \int_0^x p(x)\,\mathrm{d}x = x^{n+1},$$

and inversion is simple: $P^{-1}(x) = \sqrt[n+1]{x}$. Therefore, given a uniform random variable ξ, samples can be drawn from the power distribution as

$$X = \sqrt[n+1]{\xi}. \tag{13.4}$$

Another approach is to use a sampling trick that works only for the power distribution, selecting $X = \max(\xi_1, \xi_2, \ldots, \xi_{n+1})$. This random variable is distributed according to the power distribution as well. To see why, note that $Pr\{X < x\}$ is the probability that *all* the $\xi_i < x$. But the ξ_i are independent, so

$$Pr\{X < x\} = \prod_{i=1}^{n+1} Pr\left\{\xi_i < x\right\} = x^{n+1},$$

which is exactly the desired CDF. Depending on the speed of your random number generator, this technique can be faster than the inversion method for small values of n.

Example: Exponential Distribution

When rendering images with participating media, it is frequently useful to draw samples from a distribution $p(x) \propto \mathrm{e}^{-ax}$. As before, the first step is to normalize this distribution so that it integrates to one. In this case, the range of values x we'd like the generated samples to cover is $[0, \infty)$ rather than $[0, 1]$, so

$$\int_0^\infty c\mathrm{e}^{-ax}\,\mathrm{d}x = -\left.\frac{c}{a}\mathrm{e}^{-ax}\right|_0^\infty = \frac{c}{a} = 1.$$

Thus we know that $c = a$, and our PDF is $p(x) = a\mathrm{e}^{-ax}$. Now, we integrate to find $P(x)$:

$$P(x) = \int_0^x a\mathrm{e}^{-ax'}\,\mathrm{d}x' = 1 - \mathrm{e}^{-ax}.$$

This function is easy to invert:

$$P^{-1}(x) = -\frac{\ln(1-x)}{a},$$

and we can draw samples thusly:

$$X = -\frac{\ln(1-\xi)}{a}.$$

This equation can be further simplified by making the observation that if ξ is a uniformly distributed random number, so is $1 - \xi$, so we can replace $1 - \xi$ by ξ without changing the distribution. Therefore, our final sampling strategy is

$$X = -\frac{\ln(\xi)}{a}.$$

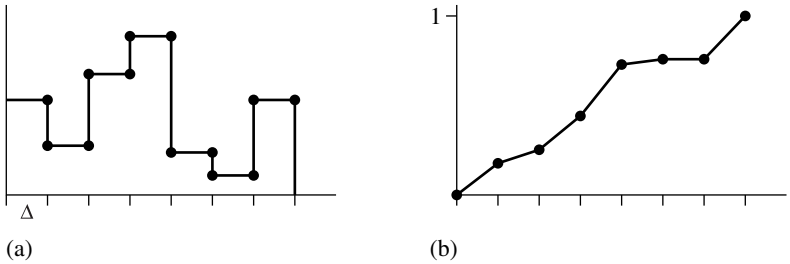


Figure 13.4: (a) Probability density function for a piecewise-constant 1D function and (b) cumulative distribution function defined by this PDF.

Example: Piecewise-Constant 1D Functions

An interesting exercise is to work out how to sample from 1D piecewise-constant functions (step functions). Without loss of generality, we will just consider piecewise-constant functions defined over $[0, 1]$.

Assume that the 1D function's domain is split into N equal-sized pieces of size $\Delta = 1/N$. These regions start and end at points $x_i = i\Delta$, where i ranges from 0 to N, inclusive. Within each region, the value of the function $f(x)$ is a constant (Figure 13.4(a)). The value of $f(x)$ is

$$f(x) = \begin{cases} v_0 & x_0 \le x < x_1 \\ v_1 & x_1 \le x < x_2 \\ \vdots \end{cases}.$$

The integral $\int f(x)\,\mathrm{d}x$ is

$$c = \int_0^1 f(x)\,\mathrm{d}x = \sum_{i=0}^{N-1} \Delta v_i = \sum_{i=0}^{N-1} \frac{v_i}{N}, \qquad [13.5]$$

and so it is easy to construct the PDF $p(x)$ for $f(x)$ as $f(x)/c$. By direct application of the relevant formulae, the CDF $P(x)$ is a piecewise linear function defined at points x_i by

$$\begin{aligned} P(x_0) &= 0 \\ P(x_1) &= \int_{x_0}^{x_1} p(x)\,\mathrm{d}x = \frac{v_0}{Nc} = P(x_0) + \frac{v_0}{Nc} \\ P(x_2) &= \int_{x_0}^{x_2} p(x)\,\mathrm{d}x = \int_{x_0}^{x_1} p(x)\,\mathrm{d}x + \int_{x_1}^{x_2} p(x)\,\mathrm{d}x = P(x_1) + \frac{v_1}{Nc} \\ P(x_i) &= P(x_{i-1}) + \frac{v_{i-1}}{Nc}. \end{aligned}$$

Between two points x_i and x_{i+1}, the CDF is linearly increasing with slope v_i/c.

Recall that in order to sample $f(x)$ we need to invert the CDF to find the value x such that

$$\xi = \int_0^x p(x')\,dx' = P(x).$$

Because the CDF is monotonically increasing, the value of x must be between the x_i and x_{i+1} such that $P(x_i) \leq \xi$ and $\xi \leq P(x_{i+1})$. Given an array of CDF values, this pair of $P(x_i)$ values can be efficiently found with a binary search.

`Distribution1D` is a small utility class that represents a piecewise-constant 1D function's PDF and CDF and provides methods to perform this sampling efficiently.

```
⟨Monte Carlo Utility Declarations⟩ +≡
    struct Distribution1D {
        ⟨Distribution1D Public Methods 648⟩
    private:
        ⟨Distribution1D Private Data 648⟩
    };
```

The `Distribution1D` constructor takes `n` values of a piecewise-constant function `f`. It makes its own copy of the function values, computes the function's CDF, and also stores the integral of the function, `funcInt`. Note that the constructor allocates `n+1` floats for the `cdf` array because if $f(x)$ has N step values, then we need to store the value of the CDF at each of the $N + 1$ values of x_i. Storing the CDF value of 1 at the end of the array is redundant but simplifies the sampling code later.

```
⟨Distribution1D Public Methods⟩ ≡ 648
    Distribution1D(const float *f, int n) {
        count = n;
        func = new float[n];
        memcpy(func, f, n*sizeof(float));
        cdf = new float[n+1];
        ⟨Compute integral of step function at x_i 648⟩
        ⟨Transform step function integral into CDF 649⟩
    }
```

```
⟨Distribution1D Private Data⟩ ≡ 648
    float *func, *cdf;
    float funcInt;
    int count;
```

This constructor computes the integral of $f(x)$ using Equation (13.5). It stores the result in the `cdf` array for now so that it doesn't need to allocate additional temporary space for it.

```
⟨Compute integral of step function at x_i⟩ ≡ 648
    cdf[0] = 0.;
    for (int i = 1; i < count+1; ++i)
        cdf[i] = cdf[i-1] + func[i-1] / n;
```

Distribution1D 648
Distribution1D::cdf 648
Distribution1D::count 648
Distribution1D::func 648

Now that the value of the integral over all of $[0, 1]$ is stored in `cdf[n]`, this value can be copied into `funcInt` and the CDF can be normalized by dividing through by it:

⟨*Transform step function integral into CDF*⟩ ≡ 648
```
funcInt = cdf[count];
for (int i = 1; i < n+1; ++i)
    cdf[i] /= funcInt;
```

The `Distribution1D::SampleContinuous()` method uses the given random sample `u` to sample from its distribution. It returns the corresponding value $x \in [0, 1)$ and the value of the PDF $p(x)$.

⟨*Distribution1D Public Methods*⟩ +≡ 648
```
float SampleContinuous(float u, float *pdf) const {
    ⟨Find surrounding CDF segments and offset 649⟩
    ⟨Compute offset along CDF segment 649⟩
    ⟨Compute PDF for sampled offset 649⟩
    ⟨Return x ∈ [0, 1) corresponding to sample 649⟩
}
```

This method first finds the pair of CDF values that straddle `u`, setting `offset` to the offset to the first of the two of them. Because the `cdf` array is monotonically increasing (and is thus sorted), we can use a binary search function provided by the C++ standard library: `lower_bound()` takes a pointer to the start of the array and a pointer one position past the end of the array as well as the value to search for. Explicitly storing the value 1 at the end of the array makes it possible to just use `lower_bound()` directly and not include additional logic to handle the end-case. In the rare (but valid) case of `u == 0`, then the `max()` test below ensures that the value -1 isn't incorrectly computed for `offset`.

⟨*Find surrounding CDF segments and* `offset`⟩ ≡ 649, 650
```
float *ptr = std::lower_bound(cdf, cdf+count+1, u);
int offset = max(0, int(ptr-cdf-1));
```

Given the pair of CDF values, we can compute x. First, we determine how far `u` is between `cdf[offset]` and `cdf[offset+1]`, `du`, where `du` is zero if `u == cdf[offset]` and goes up to one if `u == cdf[offset+1]`. Because the CDF is piecewise linear, the sample value x is the same offset between x_i and x_{i+1} (Figure 13.4(b)).

⟨*Compute offset along CDF segment*⟩ ≡ 649
```
float du = (u - cdf[offset]) / (cdf[offset+1] - cdf[offset]);
```

The PDF for this sample $p(x)$ is easily computed since we have the function's integral in `funcInt`.

Distribution1D::cdf 648
Distribution1D::count 648
Distribution1D::func 648
Distribution1D::funcInt 648
Distribution1D::SampleContinuous() 649

⟨*Compute PDF for sampled offset*⟩ ≡ 649
```
if (pdf) *pdf = func[offset] / funcInt;
```

Finally, the appropriate value of x is computed and returned.

⟨*Return x ∈ [0, 1) corresponding to sample*⟩ ≡ 649
```
return (offset + du) / count;
```

In a small overloading of semantics, `Distribution1D` can also be used for discrete 1D probability distributions where there are some number of buckets n, each with some weight, and we'd like to sample among the buckets with probability proportional to their relative weights. This functionality is used, for example, by some of the `Integrators` in Chapter 15 that compute a discrete distribution for the light sources in the scene with weights given by the lights' powers. Sampling from the discrete distribution just requires figuring out which pair of CDF values the sample value lies between; the PDF is computed as the discrete probability of sampling the corresponding bucket.

⟨*Distribution1D Public Methods*⟩ +≡ 648

```
int SampleDiscrete(float u, float *pdf) const {
    ⟨Find surrounding CDF segments and offset 649⟩
    if (pdf) *pdf = func[offset] / (funcInt * count);
    return offset;
}
```

13.3.2 THE REJECTION METHOD

For some functions $f(x)$, it may not be possible to integrate them in order to find their PDFs, or it may not be possible to analytically invert their CDFs. The *rejection method* is a technique for generating samples according to a function's distribution without needing to do either of these steps; it is essentially a dart-throwing approach. Assume that we want to draw samples from some such function $f(x)$ but we do have a PDF $p(x)$ that satisfies $f(x) < c\ p(x)$ for some scalar constant c, and suppose that we do know how to sample from p. The rejection method is then:

loop forever:
 sample X from p's distribution
 if $\xi < f(X)/(c\ p(X))$ then
 return X

This procedure repeatedly chooses a pair of random variables (X, ξ). If the point $(X, \xi\ c\ p(X))$ lies under $f(X)$, then the sample X is accepted. Otherwise, it is rejected and a new sample pair is chosen. This idea is illustrated in Figure 13.5. Without going into too much detail, it should be clear that the efficiency of this scheme depends on how tightly $c\ p(x)$ bounds $f(x)$. This technique works in any number of dimensions.

In practice, rejection sampling isn't used in any of the Monte Carlo algorithms currently implemented in `pbrt`. We will normally prefer to find distributions that are similar to $f(x)$ that can be sampled directly, for reasons that will be explained in the next chapter. Nevertheless, rejection sampling is an important technique to be aware of, particularly when debugging Monte Carlo implementations. For example, if one suspects the presence of a bug in code that draws samples from some distribution using the inversion method, then one can replace it with a straightforward implementation based on the rejection method and see if the Monte Carlo estimator computes the same result. Of course, it's necessary to take many samples in situations like these, so that variance in the estimates doesn't mask errors.

`Distribution1D` 648
`Distribution1D::count` 648
`Distribution1D::func` 648
`Distribution1D::funcInt` 648

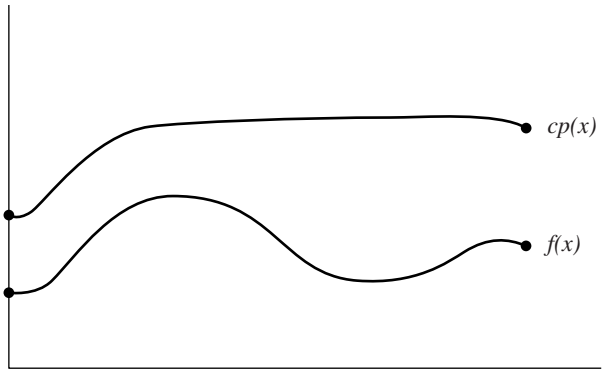


Figure 13.5: Rejection sampling generates samples according to the distribution of an arbitrary function $f(x)$ even if f's PDF is unknown or its CDF can't be inverted. If some distribution $p(x)$ and a scalar constant c are known such that $f(x) < c\,p(x)$, then samples can be drawn from $p(x)$ and randomly accepted with the rejection method. The closer the fit of $c\,p(x)$ to $f(x)$, the more efficient this process is.

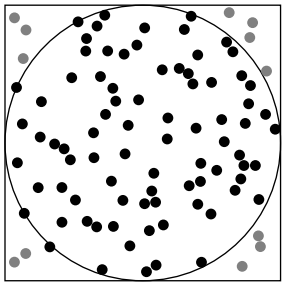


Figure 13.6: Rejection Sampling a Circle. One approach to finding uniform points in the unit circle is to sample uniform random points in the unit square and reject all that lie outside the circle. The remaining points will be uniformly distributed within the circle.

Example: Rejection Sampling a Unit Circle

Suppose we want to select a uniformly distributed point inside a unit circle. Using the rejection method, we simply select a random (x, y) position inside the circumscribed square and return it if it falls inside the circle. This process is shown in Figure 13.6.

RejectionSampleDisk() 652

The function `RejectionSampleDisk()` implements this algorithm. A similar approach will work to generate uniformly distributed samples inside for any complex shape as long as it has an inside–outside test.

⟨*Monte Carlo Function Definitions*⟩ +≡

```
void RejectionSampleDisk(float *x, float *y, RNG &rng) {
    float sx, sy;
    do {
        sx = 1.f - 2.f * rng.RandomFloat();
        sy = 1.f - 2.f * rng.RandomFloat();
    } while (sx*sx + sy*sy > 1.f);
    *x = sx;
    *y = sy;
}
```

In general, the efficiency of rejection sampling depends on the percentage of samples that are expected to be rejected. For the problem of finding uniform points in the 2D case, this is easy to compute. It is the area of the circle divided by the area of the square: $\frac{\pi}{4} \approx 78.5\%$. If the method is applied to generate samples in hyperspheres in the general n-dimensional case, however, the volume of an n-dimensional hypersphere actually goes to *zero* as n increases and this approach becomes increasingly inefficient.

⋆ 13.4 METROPOLIS SAMPLING

Metropolis sampling is a technique that has the remarkable property that it can generate a set of samples from any non-negative function f that are distributed proportionally to f's value (Metropolis et al. 1953).[6] Remarkably, it does this without requiring anything more than the ability to evaluate f; it is not necessary to be able to integrate f, normalize the integral, and invert the resulting PDF. Furthermore, *every* sample generated is from the function's PDF; Metropolis sampling doesn't share the shortcoming of rejection sampling that many candidate samples may be rejected before one is accepted. It can thus efficiently generate samples from a wider variety of functions than the techniques introduced in the previous section. It forms the foundation of the Metropolis light transport algorithm implemented in Section 15.7.

Metropolis sampling does have a few disadvantages: successive samples in the sequence are often correlated, and it is thus not possible to ensure that a small number of samples generated by Metropolis is well distributed across the domain. It's only in the limit over a large number of samples that the samples will cover the domain. As such, the variance reduction advantages of techniques like stratified sampling (Section 14.2.1) are generally not available when using Metropolis sampling.

13.4.1 BASIC ALGORITHM

More concretely, the Metropolis algorithm generates a set of samples X_i from a function f, which is defined over an arbitrary-dimensional state space Ω (frequently, $\Omega = \mathbb{R}^n$) and returns a value in the reals, $f{:}\Omega \to \mathbb{R}$. After the first sample $X_0 \in \Omega$ has been selected,

RNG 1003

RNG::RandomFloat() 1003

6 We will refer to the Monte Carlo sampling algorithm as "the Metropolis algorithm" here. Other commonly used shorthands for it include M(RT)^2, for the initials of the authors of the original paper, and Metropolis–Hastings, which gives a nod to Hastings, who generalized the technique (Fishman 1996). It is also commonly known as Markov Chain Monte Carlo.

each subsequent sample X_i is generated by using a random *mutation* to X_{i-1} to compute a proposed sample X'. The mutation may be accepted or rejected, and X_i is accordingly set to either X' or X_{i-1}. When these transitions from one state to another are chosen subject to a few requirements (to be described shortly), the distribution of X_i values that results reaches an equlibrium distribution; this distribution is the *stationary distribution*. In the limit, the distribution of the set of samples $X_i \in \Omega$ is proportional to $f(x)$'s probability density function $p(x) = f(x)/\int_\Omega f(x)\mathrm{d}\Omega$.

In order to generate the correct distribution of samples, it is necessary to generate proposed mutations and then accept or reject the mutations subject to a few constraints. Assume that we have a method of proposing mutations that proposes changing from a given state X into a proposed state X' (this might be done by perturbing X in some way, or even by generating a completely new value). We must be able to compute a tentative transition function $T(X \to X')$ that gives the probability density of the mutation technique's proposing a transition to X', given that the current state is X. (Section 13.4.2 will discuss considerations for designing transition functions.)

Given a transition function, it is possible to define an *acceptance probability* $a(X \to X')$ that gives the probability of accepting a proposed mutation from X to X' in a way that ensures that the distribution of samples is proportional to $f(x)$. If the distribution is already in equlibrium, the transition density between any two states must be equal:[7]

$$f(X)\, T(X \to X')\, a(X \to X') = f(X')\, T(X' \to X)\, a(X' \to X). \qquad [13.6]$$

This property is called *detailed balance*.

Since f and T are set, Equation (13.6) tells us how a must be defined. In particular, a definition of a that maximizes the rate at which equlibrium is reached is

$$a(X \to X') = \min\left(1, \frac{f(X')\, T(X' \to X)}{f(X)\, T(X \to X')}\right). \qquad [13.7]$$

One thing to immediately notice from Equation (13.7) is that, if the transition probability density is the same in both directions, the acceptance probability simplifies to

$$a(X \to X') = \min\left(1, \frac{f(X')}{f(X)}\right). \qquad [13.8]$$

Put together, we have the basic Metropolis sampling algorithm in pseudocode:

```
X = X0
for i = 1 to n
    X' = mutate(X)
    a = accept(X, X')
    if (random() < a)
        X = X'
    record(X)
```

7 See Kalos and Whitlock (1986) or Veach's thesis (1997) for a rigorous derivation.

This code generates n samples by mutating the previous sample and computing acceptance probabilities as in Equation (13.7). Each sample X_i can then be recorded in a data structure or used as a sample for integration.

Because the Metropolis algorithm naturally avoids parts of Ω where $f(x)$'s value is relatively low, few samples will be accumulated there. In order to get some information about $f(x)$'s behavior in such regions, the *expected values* technique can be used to enhance the basic Metropolis algorithm. In this case, we still decide which state to transition into as before, but we record a sample at each of X and X', regardless of which one is selected by the acceptance criteria. Each of these recorded samples has a weight associated with it, where the weights are the probabilities $(1 - a)$ for X and a for X', where a is the acceptance probability. Expected values doesn't change the way we decide which state, X or X', to use at the next step; that part of the computation remains the same.

Updated pseudocode shows the idea:

```
X = X0
for i = 1 to n
    X' = mutate(X)
    a = accept(X, X')
    record(X, (1-a) * weight)
    record(X', a * weight)
    if (random() < a)
        X = X'
```

Comparing the two pieces of pseudocode, we can see that in the limit, the same weight distribution will be accumulated for X and X'. Expected values more quickly give a smoother result and more information about the areas where $f(x)$ is low than the basic algorithm does.

13.4.2 CHOOSING MUTATION STRATEGIES

In general, one has a lot of freedom in choosing mutation strategies, subject to being able to compute the tenative transition density $T(X \to X')$. Recall from Equation (13.8) that if the transition densities are symmetric then it is not even necessary to be able to compute them to apply the Metropolis sampling algorithm. It's easy to apply multiple mutation strategies, so if there are some that are effective in only some circumstances it doesn't hurt to try using them as one of a set of approaches.

It is generally desirable that mutations propose large changes to the current sample rather than small ones. Doing so more quickly explores the state space, rather than letting the sampler get stuck in a small region of it. However, when the function's value $f(X)$ is relatively large at the current sample X, then it is likely that many proposed mutations will be rejected (consider the case where $f(X) \gg f(X')$ in Equation (13.8); $a(X \to X')$ will be very small). We'd like to avoid the case where many samples in a row are the same, again to better explore new parts of the state space: staying in the same state for many samples in a row leads to increased variance—intuitively, it makes sense that the more we move around Ω, the better the overall results will be. For this case, small mutations are likely to propose samples X' where f is still relatively large, leading to higher acceptance properties.

Thus, one useful mutation approach is to apply random perturbations to the current sample X. If the sample X is a vector of real numbers $(x_0, x_1, \ldots)$, then some or all of the samples x_i can be perturbed. One possibility is to perturb by adding or subtracting a scaled random variable:

$$x_i' = x_i \pm s\,\xi$$

for some scale factor s. Another useful technique is to perturb using values from an exponential distribution:

$$x_i' = x_i \pm b\,\mathrm{e}^{-\log(b/a)\,\xi}, \tag{13.9}$$

which generates an exponentially distributed sample in the range $[a, b]$. This approach can be more robust than using a single scale factor s by frequently proposing small mutations but periodically proposing very large ones. Note that this method is symmetric, so we don't need to compute the transition densities $T(X \to X')$ when using it with Metropolis sampling.

A related mutation approach is to just discard the current sample entirely and generate a new one with uniform random numbers:

$$x_i = \xi.$$

(Note that this is also a symmetric method.) Occasionally generating a completely new sample in this manner is important since it ensures that we don't get stuck in one part of the state space and never sample the rest of it. In general, it's necessary that it be possible to reach all states $X \in \Omega$ where $f(X) > 0$ with nonzero probability (this property is called *ergodicity*). In particular, to ensure ergodicity it suffices that $T(X \to X') > 0$ for all X and X' where $f(X) > 0$ and $f(X') > 0$.

Another approach is to use PDFs that match some part of the function being sampled. If we have a PDF $p(x)$ that is similar to some component of f, then we can use that to derive a mutation strategy by just drawing new samples $X \sim p$. In this case, the transition function is straightforward:

$$T(X \to X') = p(X').$$

In other words, the current state X doesn't matter for computing the transition density: we propose a transition into a state X' with a density that depends only on the newly proposed state X' and not at all on the current state.

13.4.3 START-UP BIAS

One issue that we've sidestepped thus far is how the initial sample X_0 is computed. The transition and acceptance methods above tell us how to generate new samples X_{i+1}, but all presuppose that the current sample X_i has itself *already* been sampled with probability proportional to f. Using a sample not from f's distribution leads to a problem called *start-up bias*.

A common solution to this problem is to run the Metropolis sampling algorithm for some number of iterations from an arbitrary starting state, discard the samples that are generated, and then start the process for real, assuming that that has brought us to an appropriately sampled X value. This is unsatisfying for two reasons: first, the expense of

taking the samples that were then discarded may be high, and, second, we can only guess at how many initial samples must be taken in order to remove start-up bias.

A straightforward alternative can be used if another sampling method is available: an initial value X_0 is sampled using any density function $X_0 \sim p(x)$. We start the Markov chain from the state X_0, but we weight the contributions of all of the samples that we generate by the weight

$$w = \frac{f(X_0)}{p(X_0)}.$$

This method eliminates start-up bias completely and does so in a predictable manner.

The only potential problem comes if $f(X_0) = 0$ for the X_0 we chose; in this case, all samples will have a weight of zero! This doesn't mean that the algorithm is biased, however; the expected value of the result still converges to the correct distribution (see Veach (1997) for further discussion and for a proof of the correctness). To reduce variance and avoid this risk, we can instead sample a set of N candidate sample values, $Y_1, \ldots, Y_N$, defining a weight for each by

$$w_i = \frac{f(Y_i)}{p(Y_i)}.$$

We then choose the starting X_0 sample for the Metropolis algorithm from the Y_i with probability proportional to their relative weights and compute a sample weight w as the average of all of the w_i weights. All subsequent samples X_i that are generated by the Metropolis algorithm are then weighted by the sample weight w.

13.4.4 ESTIMATING INTEGRALS WITH METROPOLIS SAMPLING

We can apply the Metropolis algorithm to estimating integrals such as $\int f(x)g(x)\,\mathrm{d}\Omega$. Doing so is the basis of the Metropolis light transport algorithm implemented in Section 15.7.

To see how the samples from Metropolis sampling can be used in this way, recall that the standard Monte Carlo estimator, Equation (13.3), says that

$$\int_\Omega f(x)g(x)\,\mathrm{d}\Omega \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)g(X_i)}{p(X_i)},$$

where X_i are sampled from a density function $p(x)$. Thus, if we apply Metropolis sampling and generate a set of samples, $X_1, \ldots, X_N$, from a density function that is proportional to $f(x)$, then we can estimate this integral as

$$\int_\Omega f(x)g(x)\,\mathrm{d}\Omega \approx \left[\frac{1}{N}\sum_{i=1}^{N} g(X_i)\right] \cdot \int_\Omega f(x)\,\mathrm{d}\Omega. \qquad [13.10]$$

13.4.5 EXAMPLE: ONE-DIMENSIONAL SETTING

In order to illustrate some of the ideas in this section, we'll show how Metropolis sampling can be used to sample a simple one-dimensional function, defined over $\Omega = [0, 1]$

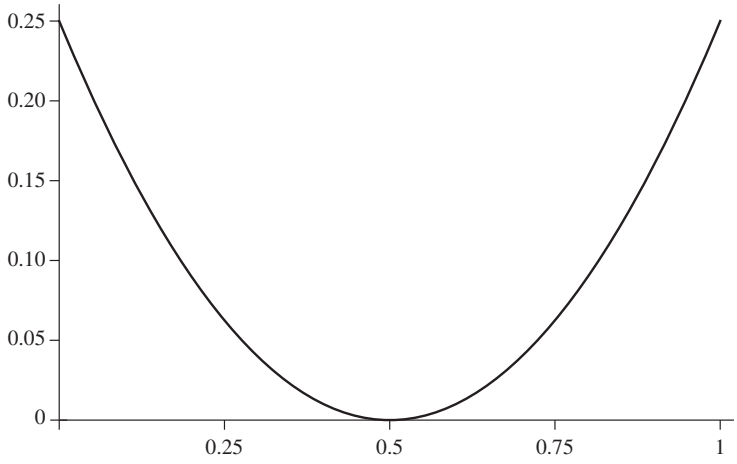


Figure 13.7: Graph of the function $f(x)$ used to illustrate Metropolis sampling in this section.

and zero everywhere else (see Figure 13.7).

$$f(x) = \begin{cases} (x - .5)^2 & 0 \leq x \leq 1 \\ 0 & \text{otherwise.} \end{cases} \quad [13.11]$$

For this example, assume that we don't actually know the exact form of f—it's just a black box that we can evaluate at particular x values. (Clearly, if we knew that f was just Equation (13.11), there'd be no need for Metropolis in order to draw samples from its distribution!)

We'll define two mutation strategies based on the ideas introduced in Section 13.4.2 and randomly choose among them each time a mutation is proposed, according to a desired distribution of how frequently each is to be used.

The first strategy, mutate_1, discards the current sample X and uniformly samples a new one, X', from the entire state space $[0, 1]$. The transition function for this mutation is straightforward. For mutate_1, since we are uniformly sampling over $[0, 1]$, the probability density is uniform over the entire domain; in this case, the density is just one everywhere. We have

$$\text{mutate}_1(X) \rightarrow \xi$$
$$T_1(X \rightarrow X') = 1.$$

The second mutation adds a random offset between $\pm.05$ to the current sample X in an effort to sample repeatedly in the parts of f that make a high contribution to the overall distribution. The transition probability density is zero if X and X' are far enough away that mutate_2 will never mutate from one to the other; otherwise, the density is constant. Normalizing the density so that it integrates to one over its domain gives the value $1/0.1$. Both this and mutate_1 are symmetric, so the transition densities aren't needed to

implement the sampling algorithm.

$$\text{mutate}_2(X) \rightarrow X + .1(\xi - .5)$$

$$T_2(X \rightarrow X') = \begin{cases} \frac{1}{0.1} & |X - X'| \leq .05 \\ 0 & \text{otherwise.} \end{cases}$$

To find the initial sample, we only need to take a single sample with a uniform PDF over Ω, since $f(x) > 0$ except for a single point in Ω for which there is zero probability of sampling:

$$X_0 = \xi.$$

The sample weight w is then just $f(X_0)$.

We can now run the Metropolis algorithm and generate samples X_i of f. At each transition, we have two weighted samples to record (recall the expected values pseudocode from Section 13.4.1). A simple approach for reconstructing the approximation to f's probability distribution is to store sums of the weights in a set of buckets of uniform width; each sample falls in a single bucket and contributes to it. Figure 13.8 shows some results. For both graphs, a chain of 10,000 mutations was followed, with the sample weights accumulated in 50 buckets over $[0, 1]$.

On the left graph, only mutate_1 was used. This alone isn't a very effective mutation, since it doesn't take advantage of the times when it has found a sample in a region of Ω where f has a relatively large value to generate additional samples in that neighborhood. However, the graph does suggest that the algorithm is converging to the correct distribution.

On the right, one of mutate_1 and mutate_2 was randomly chosen, with probabilities of 10% and 90%, respectively. We see that for the same number of samples taken, we converge to f's distribution with less variance. This is because the algorithm is more effectively able to concentrate its work in areas where f's value is large, proposing fewer mutations to parts of state space where f's value is low. For example, if $X = .8$ and the second mutation proposes $X' = .75$, this will be accepted $f(.75)/f(.8) \approx 69\%$ of the time, while mutations from .75 to .8 will be accepted $\min(1, 1.44) = 100\%$ of the time. Thus, we see how the algorithm naturally tends to try to avoid spending time sampling around the dip in the middle of the curve.

One important thing to note about these graphs is that the y axis has units that are different than those in Figure 13.7, where f is graphed. Recall that Metropolis sampling provides a set of samples distributed according to f's probability density; as such (for example), we would get the same sample distribution for another function $g = 2f$. If we wish to reconstruct an approximation to f directly from Metropolis samples, we must compute a normalization factor and use it to scale the PDF.

Figure 13.9 shows the surprising result of only using mutate_2 to propose sample values. On the left, 10,000 samples were taken using just that mutation. Clearly, things have gone awry—*no* samples $X_i > .5$ were generated and the result doesn't bear much resemblance to f.

Thinking about the acceptance probability again, we can see that it would take a large number of mutations, each with low probability of acceptance, to move X_i down close

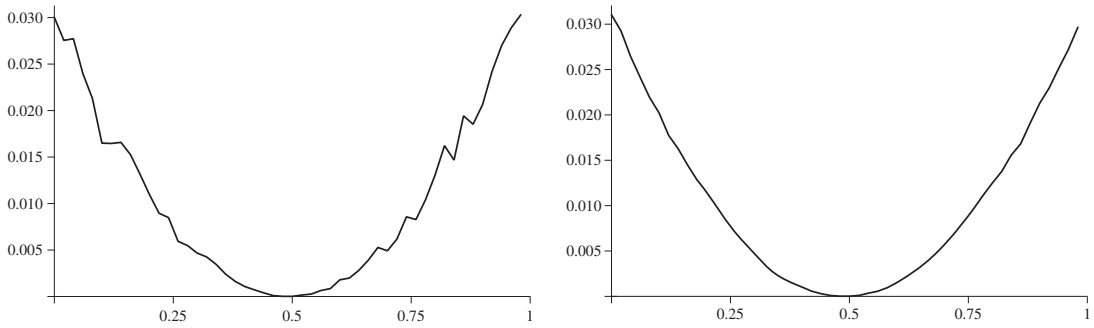


Figure 13.8: Comparison of Metropolis Sampling Strategies. (a) Only mutate_1 was used, randomly selecting a completely new value at each step. (b) Both mutate_1 and mutate_2 were used, with a 1:9 ratio. For the same number of samples, variance is substantially lower, thanks to a higher rate of acceptance of mutate_2's proposed mutations.

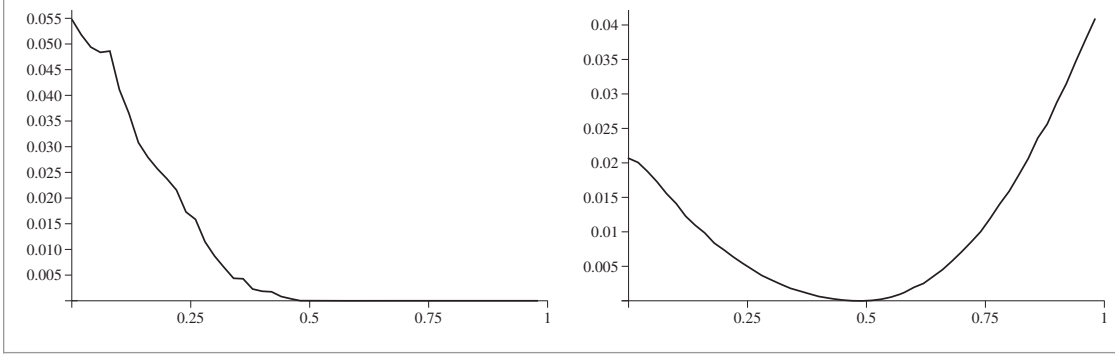


Figure 13.9: Why it is important to periodically pick a completely new sample value with Metropolis sampling. (a) 10,000 iterations using only mutate_2 were computed, (b) 300,000 iterations. It is very unlikely that a series of mutations will be able to move from one side of the curve, across .5, to the other side, since mutations that move toward areas where f's value is low will usually be rejected. As such, the results are inaccurate for these numbers of iterations. (It's small solace that they would be correct in the limit.)

enough to .5 such that mutate_2's short span would be enough to move over to the other side. Since the Metropolis algorithm tends to stay away from the lower-valued regions of f (recall the comparison of probabilities for moving from .8 to .75 versus moving from .75 to .8), this happens quite rarely. The right side of Figure 13.9 shows what happens if 300,000 samples are taken. This was enough to be able to jump from one side of .5 to the other a few times, but not enough to get close to the correct distribution. Using mutate_2 alone is thus not mathematically incorrect, just inefficient: it does have nonzero probability of proposing a transition from any state to any other state (through a chain of multiple transitions).

13.5 TRANSFORMING BETWEEN DISTRIBUTIONS

In describing the inversion method, we introduced a technique that generates samples according to some distribution by transforming canonical uniform random variables in a particular manner. Here, we will investigate the more general question of which distribution results when we transform samples from an arbitrary distribution to some other distribution with a function f.

Suppose we are given random variables X_i that are already drawn from some PDF $p_x(x)$. Now, if we compute $Y_i = y(X_i)$, we would like to find the distribution of the new random variable Y_i. This may seem like an esoteric problem, but we will see that understanding this kind of transformation is critical for drawing samples from multidimensional distribution functions.

The function $y(x)$ must be a one-to-one transformation; if multiple values of x mapped to the same y value, then it would be impossible to unambiguously describe the probability density of a particular y value. A direct consequence of y being one-to-one is that its derivative must either be strictly greater than zero or strictly less than zero, which implies that

$$Pr\{Y \leq y(x)\} = Pr\{X \leq x\},$$

and therefore

$$P_y(y) = P_y(y(x)) = P_x(x).$$

This relationship between CDFs leads directly to the relationship between their PDFs. If we assume that y's derivative is greater than zero, differentiating gives

$$p_y(y)\frac{\mathrm{d}y}{\mathrm{d}x} = p_x(x),$$

and so

$$p_y(y) = \left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)^{-1} p_x(x).$$

In general, y's derivative is either strictly positive or strictly negative, and the relationship between the densities is

$$p_y(y) = \left|\frac{\mathrm{d}y}{\mathrm{d}x}\right|^{-1} p_x(x).$$

How can we use this formula? Suppose that $p_x(x) = 2x$ over the domain $[0, 1]$, and let $Y = \sin X$. What is the PDF of the random variable Y? Because we know that $\mathrm{d}y/\mathrm{d}x = \cos x$,

$$p_y(y) = \frac{p_x(x)}{|\cos x|} = \frac{2x}{\cos x} = \frac{2 \arcsin y}{\sqrt{1 - y^2}}.$$

This procedure may seem backwards—usually we have some PDF that we want to sample from, not a given transformation. For example, we might have X drawn from some

$p_x(x)$ and would like to compute Y from some distribution $p_y(y)$. What transformation should we use? All we need is for the CDFs to be equal, or $P_y(y) = P_x(x)$, which immediately gives the transformation

$$y(x) = P_y^{-1}\left(P_x(x)\right).$$

This is a generalization of the inversion method, since if X were uniformly distributed over $[0, 1]$ then $P_x(x) = x$, and we have the same procedure as was introduced previously.

13.5.1 TRANSFORMATION IN MULTIPLE DIMENSIONS

In the general n-dimensional case, a similar derivation gives the analogous relationship between different densities. We will not show the derivation here; it follows the same form as the one-dimensional case. Suppose we have an n-dimensional random variable X with density function $p_x(x)$. Now let $Y = T(X)$, where T is a bijection. In this case, the densities are related by

$$p_y(y) = p_y(T(x)) = \frac{p_x(x)}{|J_T(x)|},$$

where $|J_T|$ is the absolute value of the determinant of T's Jacobian matrix, which is

$$\begin{pmatrix} \partial T_1/\partial x_1 & \cdots & \partial T_1/\partial x_n \\ \vdots & \ddots & \vdots \\ \partial T_n/\partial x_1 & \cdots & \partial T_n/\partial x_n \end{pmatrix},$$

where T_i are defined by $T(x) = (T_1(x), \ldots, T_n(x))$.

13.5.2 EXAMPLE: POLAR COORDINATES

The polar transformation is given by

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta. \end{aligned}$$

Suppose we draw samples from some density $p(r, \theta)$. What is the corresponding density $p(x, y)$? The Jacobian of this transformation is

$$J_T = \begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial \theta} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial \theta} \end{pmatrix} = \begin{pmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{pmatrix},$$

and the determinant is $r\left(\cos^2\theta + \sin^2\theta\right) = r$. So $p(x, y) = p(r, \theta)/r$. Of course, this is backwards from what we usually want—typically we start with a sampling strategy in Cartesian coordinates and want to transform it to one in polar coordinates. In that case, we would have

$$p(r, \theta) = r\ p(x, y).$$

13.5.3 EXAMPLE: SPHERICAL COORDINATES

Given the spherical coordinate representation of directions,

$$\begin{aligned} x &= r\sin\theta\cos\phi \\ y &= r\sin\theta\sin\phi \\ z &= r\cos\theta, \end{aligned}$$

the Jacobian of this transformation has determinant $|J_T| = r^2 \sin\theta$, so the corresponding density function is

$$p(r, \theta, \phi) = r^2 \sin\theta \; p(x, y, z).$$

This transformation is important since it helps us represent directions as points (x, y, z) on the unit sphere. Remember that solid angle is defined as the area of a set of points on the unit sphere. In spherical coordinates, we previously derived

$$d\omega = \sin\theta \; \mathrm{d}\theta \, \mathrm{d}\phi.$$

So if we have a density function defined over a solid angle Ω, this means that

$$Pr\left\{\omega \in \Omega\right\} = \int_\Omega p(\omega) \, \mathrm{d}\omega.$$

The density with respect to θ and ϕ can therefore be derived:

$$\begin{aligned} p(\theta, \phi) \, \mathrm{d}\theta \, \mathrm{d}\phi &= p(\omega) \, \mathrm{d}\omega \\ p(\theta, \phi) &= \sin\theta \; p(\omega). \end{aligned}$$

13.6 2D SAMPLING WITH MULTIDIMENSIONAL TRANSFORMATIONS

Suppose we have a 2D joint density function $p(x, y)$ that we wish to draw samples (X, Y) from. Sometimes multidimensional densities are separable and can be expressed as the product of 1D densities, for example,

$$p(x, y) = p_x(x) p_y(y),$$

for some p_x and p_y. In this case, random variables (X, Y) can be found by independently sampling X from p_x and Y from p_y. Many useful densities aren't separable, however, so we will introduce the theory of how to sample from multidimensional distributions in the general case.

Given a 2D density function, the *marginal density function* $p(x)$ is obtained by "integrating out" one of the dimensions:

$$p(x) = \int p(x, y) \, \mathrm{d}y. \tag{13.12}$$

This can be thought of as the density function for X alone. More precisely, it is the average density for a particular x over *all* possible y values.

The *conditional density function* $p(y|x)$ is the density function for y given that some particular x has been chosen (it is read "p of y given x"):

$$p(y|x) = \frac{p(x, y)}{p(x)}. \tag{13.13}$$

The basic idea for 2D sampling from joint distributions is to first compute the marginal density to isolate one particular variable and draw a sample from that density using standard 1D techniques. Once that sample is drawn, one can then compute the conditional

density function given that value and draw a sample from that distribution, again using standard 1D sampling techniques.

13.6.1 EXAMPLE: UNIFORMLY SAMPLING A HEMISPHERE

As an example, consider the task of choosing a direction on the hemisphere uniformly with respect to solid angle. Remember that a uniform distribution means that the density function is a constant, so we know that $p(\omega) = c$. In conjunction with the fact that the density function must integrate to one over its domain, we have

$$\int_{\mathcal{H}^2} p(\omega)\, \mathrm{d}\omega = 1 \Rightarrow c \int_{\mathcal{H}^2} \mathrm{d}\omega = 1 \Rightarrow c = \frac{1}{2\pi}.$$

This tells us that $p(\omega) = 1/(2\pi)$, or $p(\theta, \phi) = \sin\theta/(2\pi)$ (using a result from the previous example about spherical coordinates). Note that this density function is separable. Nevertheless, we will use the marginal and conditional densities to illustrate the multidimensional sampling technique.

Consider sampling θ first. To do so, we need θ's marginal density function $p(\theta)$:

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi)\, \mathrm{d}\phi = \int_0^{2\pi} \frac{\sin\theta}{2\pi}\, \mathrm{d}\phi = \sin\theta.$$

Now, compute the conditional density for ϕ:

$$p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{1}{2\pi}.$$

Notice that the density function for ϕ is itself uniform; this should make intuitive sense given the symmetry of the hemisphere. Now, we use the 1D inversion technique to sample each of these PDFs in turn:

$$P(\theta) = \int_0^{\theta} \sin\theta'\, \mathrm{d}\theta' = 1 - \cos\theta$$

$$P(\phi|\theta) = \int_0^{\phi} \frac{1}{2\pi}\, \mathrm{d}\phi' = \frac{\phi}{2\pi}.$$

Inverting these functions is straightforward, and again noting that we can safely replace $1 - \xi$ with ξ since these are both uniformly distributed random numbers over $[0, 1]$, we get

$$\theta = \cos^{-1}\xi_1$$
$$\phi = 2\pi\xi_2.$$

Converting these back to Cartesian coordinates, we get the final sampling formulae:

$$x = \sin\theta\cos\phi = \cos\left(2\pi\xi_2\right)\sqrt{1-\xi_1^2}$$
$$y = \sin\theta\sin\phi = \sin\left(2\pi\xi_2\right)\sqrt{1-\xi_1^2}$$
$$z = \cos\theta = \xi_1.$$

This sampling strategy is implemented in the following code. Two uniform random numbers u1 and u2 are passed in, and a vector on the hemisphere is returned.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector UniformSampleHemisphere(float u1, float u2) {
    float z = u1;
    float r = sqrtf(max(0.f, 1.f - z*z));
    float phi = 2 * M_PI * u2;
    float x = r * cosf(phi);
    float y = r * sinf(phi);
    return Vector(x, y, z);
}
```

For each sampling routine like this in pbrt, there is a corresponding function that returns the value of the PDF for a particular sample. For such functions, it is important to be clear which PDF is being evaluated—for example, for a direction on the hemisphere, we have already seen these densities expressed differently in terms of solid angle and in terms of (θ, ϕ). For hemispheres (and all other directional sampling), these functions return values with respect to solid angle. For the hemisphere, the solid angle PDF is a constant $p(\omega) = 1/(2\pi)$.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float UniformHemispherePdf() {
    return INV_TWOPI;
}
```

Sampling the full sphere uniformly over its area follows almost exactly the same derivation, which we omit here. The end result is

$$\begin{aligned} x &= \cos(2\pi\xi_2)\sqrt{1-z^2} = \cos(2\pi\xi_2)2\sqrt{\xi_1(1-\xi_1)} \\ y &= \sin(2\pi\xi_2)\sqrt{1-z^2} = \sin(2\pi\xi_2)2\sqrt{\xi_1(1-\xi_1)} \\ z &= 1-2\xi_1. \end{aligned}$$

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector UniformSampleSphere(float u1, float u2) {
    float z = 1.f - 2.f * u1;
    float r = sqrtf(max(0.f, 1.f - z*z));
    float phi = 2.f * M_PI * u2;
    float x = r * cosf(phi);
    float y = r * sinf(phi);
    return Vector(x, y, z);
}
```

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float UniformSpherePdf() {
    return 1.f / (4.f * M_PI);
}
```

INV_TWOPI 1002
M_PI 1002
Vector 57

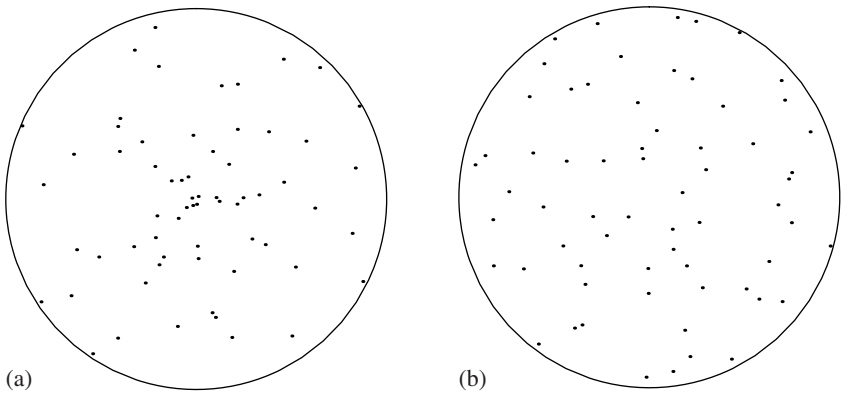


Figure 13.10: (a) When the obvious but incorrect mapping of uniform random variables to points on the disk is used, the resulting distribution is not uniform and the samples are more likely to be near the center of the disk. (b) The correct mapping gives a uniform distribution of points.

13.6.2 EXAMPLE: SAMPLING A UNIT DISK

Although the disk seems a simpler shape to sample than the hemisphere, it can be more tricky to sample uniformly because it has an incorrect intuitive solution. The wrong approach is the seemingly obvious one: $r = \xi_1, \theta = 2\pi\xi_2$. Although the resulting point is both random and inside the circle, it is *not* uniformly distributed; it actually clumps samples near the center of the circle. Figure 13.10(a) shows a plot of samples on the unit disk when this mapping was used for a set of uniform random samples (ξ_1, ξ_2). Figure 13.10(b) shows uniformly distributed samples resulting from the following correct approach.

Since we're going to sample uniformly with respect to area, the PDF $p(x, y)$ must be a constant. By the normalization constraint, $p(x, y) = 1/\pi$. If we transform into polar coordinates (see the example in Section 13.5.2), we have $p(r, \theta) = r/\pi$. Now we compute the marginal and conditional densities as before:

$$p(r) = \int_0^{2\pi} p(r, \theta)\, \mathrm{d}\theta = 2r$$
$$p(\theta|r) = \frac{p(r, \theta)}{p(r)} = \frac{1}{2\pi}.$$

As with the hemisphere case, the fact that $p(\theta|r)$ is a constant should make sense because of the symmetry of the circle. Integrating and inverting to find $P(r)$, $P^{-1}(r)$, $P(\theta)$, and $P^{-1}(\theta)$, we can find that the correct solution to generate uniformly distributed samples on a disk is

$$r = \sqrt{\xi_1}$$
$$\theta = 2\pi\xi_2.$$

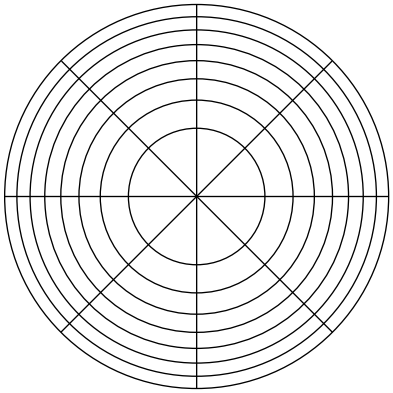


Figure 13.11: The mapping from 2D random samples to points on the disk implemented in `UniformSampleDisk()` distorts areas substantially. Each section of the disk here has equal area and represents $\frac{1}{8}$ of the unit square of uniform random samples in each direction. In general, we'd prefer a mapping that did a better job at mapping nearby (ξ_1, ξ_2) values to nearby points on the disk.

Taking the square root of ξ_1 effectively pushes the samples back toward the edge of the disk, counteracting the clumping referred to earlier.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
void UniformSampleDisk(float u1, float u2, float *x, float *y) {
    float r = sqrtf(u1);
    float theta = 2.0f * M_PI * u2;
    *x = r * cosf(theta);
    *y = r * sinf(theta);
}
```

Although this mapping solves the problem at hand, it distorts areas on the disk; areas on the unit square are elongated and/or compressed when mapped to the disk (Figure 13.11). (Section 14.2.3 will discuss in more detail why this distortion is a disadvantage.) Peter Shirley (1997) developed a "concentric" mapping from the unit square to the unit circle that avoids this problem. The concentric mapping takes points in the square $[-1, 1]^2$ to the unit disk by uniformly mapping concentric squares to concentric circles (Figure 13.12).

The mapping turns wedges of the square into slices of the disk. For example, points in the shaded area of the square in Figure 13.12 are mapped to (r, θ) by

$$r = x$$
$$\theta = \frac{y}{x}.$$

See Figure 13.13. The other four quadrants are handled analogously.

M_PI 1002
UniformSampleDisk() 666

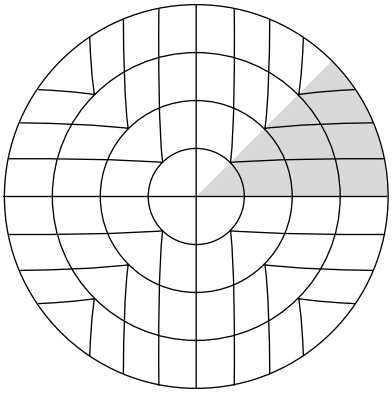


Figure 13.12: The concentric mapping maps squares to circles, giving a less distorted mapping than the first method shown for uniformly sampling points on the unit disk.

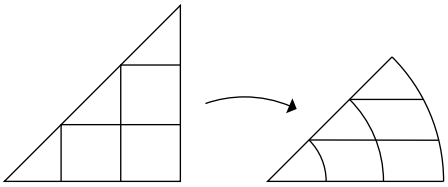


Figure 13.13: Triangular wedges of the square are mapped into (r, θ) pairs in pie-shaped slices of the circle.

⟨*Monte Carlo Function Definitions*⟩ +≡

```
void ConcentricSampleDisk(float u1, float u2, float *dx, float *dy) {
    float r, theta;
    ⟨Map uniform random numbers to [−1, 1]^2 667⟩
    ⟨Map square to (r, θ) 668⟩
    *dx = r * cosf(theta);
    *dy = r * sinf(theta);
}
```

⟨*Map uniform random numbers to* $[-1, 1]^2$⟩ ≡ 667

```
float sx = 2 * u1 - 1;
float sy = 2 * u2 - 1;
```

⟨*Map square to* (r, θ)⟩ ≡ 667
```
⟨Handle degeneracy at the origin⟩
if (sx >= -sy) {
    if (sx > sy) {
        ⟨Handle first region of disk 668⟩
    }
    else {
        ⟨Handle second region of disk⟩
    }
}
else {
    if (sx <= sy) {
        ⟨Handle third region of disk⟩
    }
    else {
        ⟨Handle fourth region of disk⟩
    }
}
theta *= M_PI / 4.f;
```

⟨*Handle first region of disk*⟩ ≡ 668
```
r = sx;
if (sy > 0.0) theta = sy/r;
else          theta = 8.0f + sy/r;
```

The remaining cases are analogous and are omitted.

13.6.3 EXAMPLE: COSINE-WEIGHTED HEMISPHERE SAMPLING

As we will see later in the next chapter, it is often useful to sample from a distribution that has a shape similar to that of the integrand being estimated. For example, because the scattering equation weights the product of the BSDF and the incident radiance with a cosine term, it is useful to have a method that generates directions that are more likely to be close to the top of the hemisphere, where the cosine term has a large value, than the bottom, where the cosine term is small.

Mathematically, this means that we would like to sample directions ω from a PDF

$$p(\omega) \propto \cos\theta.$$

Normalizing as usual,

$$\int_{\mathcal{H}^2} c\ p(\omega)\ \mathrm{d}\omega = 1$$

$$\int_0^{2\pi} \int_0^{\frac{\pi}{2}} c \cos\theta \sin\theta\ \mathrm{d}\theta\ \mathrm{d}\phi = 1$$

$$c\ 2\pi \int_0^{\pi/2} \cos\theta \sin\theta\ \mathrm{d}\theta = 1$$

M_PI 1002

$$c = \frac{1}{\pi}$$

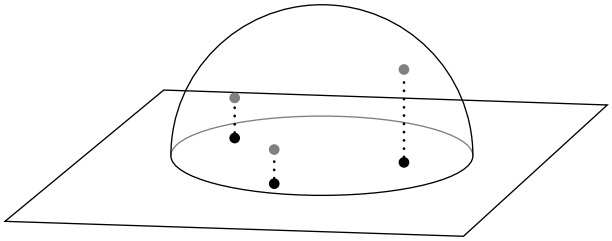


Figure 13.14: Malley's Method. To sample direction vectors from a cosine-weighted distribution, uniformly sample points on the unit disk and project them up to the unit sphere.

so

$$p(\theta, \phi) = \frac{1}{\pi} \cos\theta \sin\theta.$$

We could compute the marginal and conditional densities as before, but instead we can use a technique known as *Malley's method* to generate these cosine-weighted points. The idea behind Malley's method is that if we choose points uniformly from the unit disk and then generate directions by projecting the points on the disk up to the hemisphere above it, the resulting distribution of directions will have a cosine distribution (Figure 13.14).

Why does this work? Let (r, ϕ) be the polar coordinates of the point chosen on the disk (note that we're using ϕ instead of the usual θ here). From our calculations before, we know that the joint density $p(r, \phi) = r/\pi$ gives the density of a point sampled on the disk.

Now, we map this to the hemisphere. The vertical projection gives $\sin\theta = r$, which is easily seen from Figure 13.14. To complete the $(r, \phi) \rightarrow (\sin\theta, \phi)$ transformation, we need the determinant of the Jacobian

$$|J_T| = \begin{vmatrix} \cos\theta & 0 \\ 0 & 1 \end{vmatrix} = \cos\theta.$$

Therefore, $p(\theta, \phi) = |J_T| p(r, \phi) = \cos\theta r/\pi = (\cos\theta \sin\theta)/\pi$, which is exactly what we wanted! We have used the transformation method to prove that Malley's method generates directions with a cosine-weighted distribution. Note that this technique works regardless of the method used to sample points from the circle, so we can use Shirley's concentric mapping just as well as the simpler $(r, \theta) = (\sqrt{\xi_1}, 2\pi\xi_2)$ method.

⟨*Monte Carlo Utility Declarations*⟩ +≡

```
inline Vector CosineSampleHemisphere(float u1, float u2) {
    Vector ret;
    ConcentricSampleDisk(u1, u2, &ret.x, &ret.y);
    ret.z = sqrtf(max(0.f, 1.f - ret.x*ret.x - ret.y*ret.y));
    return ret;
}
```

ConcentricSampleDisk() 667

Vector 57

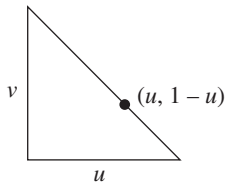


Figure 13.15: Sampling an Isosceles Right Triangle. Note that the equation of the hypotenuse is $v = 1 - u$.

Remember that all of the PDF evaluation routines in `pbrt` are defined with respect to solid angle, not spherical coordinates, so the PDF function returns a weight of $\cos\theta/\pi$.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline float CosineHemispherePdf(float costheta, float phi) {
    return costheta * INV_PI;
}
```

13.6.4 EXAMPLE: SAMPLING A TRIANGLE

Although uniformly sampling a triangle may seem like a simple task, it turns out to be more complex than the ones we've seen so far.[8] To simplify the problem, we will assume we are sampling an isosceles right triangle of area $\frac{1}{2}$. The output of the sampling routine that we will derive will be barycentric coordinates, however, so the technique will actually work for any triangle despite this simplification. Figure 13.15 shows the shape to be sampled.

We will denote the two barycentric coordinates here by (u, v). Since we are sampling with respect to area, we know that the PDF $p(u, v)$ must be a constant equal to the reciprocal of the shape's area, $\frac{1}{2}$, so $p(u, v) = 2$.

First, we find the marginal density $p(u)$:

$$p(u) = \int_0^{1-u} p(u, v)\, \mathrm{d}v = 2 \int_0^{1-u} \mathrm{d}v = 2(1 - u),$$

and the conditional density $p(v|u)$:

$$p(v|u) = \frac{p(u, v)}{p(u)} = \frac{2}{2(1-u)} = \frac{1}{1-u}.$$

The CDFs are, as always, found by integration:

8 It is possible to generate the right distribution in a triangle by sampling the enclosing parallelogram and reflecting samples on the wrong side of the diagonal back into the triangle. Although this technique is simpler than the one presented here, it is undesirable since it effectively folds the 2D uniform random samples back on top of each other—two samples that are very far away (e.g., (.01, .01) and (.99, .99)) can map to the same point on the triangle. This thwarts variance reduction techniques like stratified sampling that generate sets of well-distributed (ξ_1, ξ_2) samples and expect that they will map to well-distributed points on the object being sampled; see Section 14.2.1 for further discussion.

INV_PI 1002

$$P(u) = \int_0^u p(u')\,\mathrm{d}u' = 2u - u^2$$
$$P(v) = \int_0^v p(v'|u)\,\mathrm{d}v' = \frac{v}{1-u}.$$

Inverting these functions and assigning them to uniform random variables gives the final sampling strategy:

$$u = 1 - \sqrt{\xi_1}$$
$$v = \xi_2\sqrt{\xi_1}.$$

Notice that the two variables in this case are *not* independent!

⟨*Monte Carlo Function Definitions*⟩ +≡
```
void UniformSampleTriangle(float u1, float u2, float *u, float *v) {
    float su1 = sqrtf(u1);
    *u = 1.f - su1;
    *v = u2 * su1;
}
```

We won't provide a PDF evaluation function for this sampling strategy since the appropriate value depends on the triangle's area.

13.6.5 EXAMPLE: PIECEWISE-CONSTANT 2D DISTRIBUTIONS

Our final example will show how to sample from discrete 2D distributions. We will consider the case of a 2D function defined over $(u, v) \in [0, 1]^2$ by a 2D array of $n_u \times n_v$ sample values. This case is particularly useful for generating samples from distributions defined by texture maps and environment maps.

Consider a 2D function $f(u, v)$ defined by a set of $n_u \times n_v$ values $f[u_i, v_j]$ where $u_i \in [0, 1, \ldots, n_u - 1]$, $v_j \in [0, 1, \ldots, n_v - 1]$, and $f[u_i, v_j]$ give the constant value of f over the range $[i/n_u, (i+1)/n_u) \times [j/n_v, (j+1)/n_v)$. Given continuous values (u, v), we will use $(\tilde{u}, \tilde{v})$ to denote the corresponding discrete (u_i, v_j) indices, with $\tilde{u} = \lfloor n_u u \rfloor$ and $\tilde{v} = \lfloor n_v v \rfloor$ so that $f(u, v) = f[\tilde{u}, \tilde{v}]$.

Integrals of f are simple sums of $f[u_i, v_j]$ values, so that, for example, the integral of f over the domain is

$$I_f = \iint f(u, v)\,\mathrm{d}u\,\mathrm{d}v = \frac{1}{n_u n_v}\sum_{i=0}^{n_u-1}\sum_{j=0}^{n_v-1} f[u_i, v_j].$$

Using the definition of the PDF and the integral of f, we can find f's PDF,

$$p(u, v) = \frac{f(u, v)}{\iint f(u, v)\,\mathrm{d}u\,\mathrm{d}v} = \frac{f[\tilde{u}, \tilde{v}]}{1/(n_u n_v)\sum_i \sum_j f[u_i, v_j]}.$$

Recalling Equation (13.12), the marginal density $p(v)$ can be computed as a sum of $f[u_i, v_j]$ values

$$p(v) = \int p(u, v)\,\mathrm{d}u = \frac{(1/n_u)\sum_i f[u_i, \tilde{v}]}{I_f}. \qquad [13.14]$$

Because this function only depends on $\tilde{v}$, it is thus itself a piecewise constant 1D function, $p[\tilde{v}]$, defined by n_v values. The 1D sampling machinery from Section 13.3.1 can be applied to sampling from its distribution.

Given a v sample, the conditional density $p(u|v)$ is then

$$p(u|v) = \frac{p(u, v)}{p(v)} = \frac{f[\tilde{u}, \tilde{v}]/I_f}{p[\tilde{v}]}. \tag{13.15}$$

Note that, given a particular value of $\tilde{v}$, $p[\tilde{u}|\tilde{v}]$ is a piecewise-constant 1D function of $\tilde{u}$, that can be sampled with the usual one-dimensional approach. There are n_v such distinct 1D conditional densities, one for each possible value of $\tilde{v}$.

Putting this all together, the `Distribution2D` structure provides functionality similar to `Distribution1D`, except that it generates samples from piecewise-constant 2D distributions.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
struct Distribution2D {
    ⟨Distribution2D Public Methods 673⟩
private:
    ⟨Distribution2D Private Data 672⟩
};
```

The constructor has two tasks. First, it computes a 1D conditional sampling density $p[\tilde{u}|\tilde{v}]$ for each of the n_v individual $\tilde{v}$ values using Equation (13.15). It then computes the marginal sampling density $p[\tilde{v}]$ with Equation (13.14).

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Distribution2D::Distribution2D(const float *func, int nu, int nv) {
    for (int v = 0; v < nv; ++v) {
        ⟨Compute conditional sampling distribution for ṽ 672⟩
    }
    ⟨Compute marginal sampling distribution p[ṽ] 673⟩
}
```

First the $p[\tilde{u}|\tilde{v}]$ distributions are computed. `Distribution1D` can do this directly by being given a pointer to each of the n_v rows of n_u function values, since they're laid out linearly in memory. The I_f and $p[\tilde{v}]$ terms from Equation (13.15) don't need to be included in the values passed to `Distribution1D`, since they have the same value for all of the n_u values and are thus just a constant scale that doesn't affect the normalized distribution that `Distribution1D` computes.

⟨*Compute conditional sampling distribution for ṽ*⟩ ≡ 672
```
pConditionalV.push_back(new Distribution1D(&func[v*nu], nu));
```

⟨*Distribution2D Private Data*⟩ ≡ 672
```
vector<Distribution1D *> pConditionalV;
```

Given the conditional densities for each $\tilde{v}$ value, we can find the 1D marginal density for sampling each $\tilde{v}$ value, $p[\tilde{v}]$. The `Distribution1D` class stores the integral of the

Distribution1D 648
Distribution2D 672
Distribution2D::
pConditionalV 672

piecewise-constant function it represents in its `funcInt` member variable, so it's just necessary to copy these values to the `marginalFunc` buffer so they're stored linearly in memory for the `Distribution1D` constructor.

⟨*Compute marginal sampling distribution* $p[\tilde{v}]$⟩ ≡ 672
```
vector<float> marginalFunc;
for (int v = 0; v < nv; ++v)
    marginalFunc.push_back(pConditionalV[v]->funcInt);
pMarginal = new Distribution1D(&marginalFunc[0], nv);
```

⟨*Distribution2D Private Data*⟩ +≡ 672
```
Distribution1D *pMarginal;
```

As described previously, in order to sample from the 2D distribution, first a sample is drawn from the $p[\tilde{v}]$ marginal distribution in order to find the v coordinate of the sample. This floating-point value can be converted to the integer $\tilde{v}$ value in order to determine which of the precomputed condititional distributions should be used for sampling the u value. Figure 13.16 illustrates this idea using a low-resolution image as an example.

⟨*Distribution2D Public Methods*⟩ ≡ 672
```
void SampleContinuous(float u0, float u1, float uv[2],
                      float *pdf) const {
    float pdfs[2];
    uv[1] = pMarginal->SampleContinuous(u1, &pdfs[1]);
    int v = Clamp(Float2Int(uv[1] * pMarginal->count), 0,
                  pMarginal->count-1);
    uv[0] = pConditionalV[v]->SampleContinuous(u0, &pdfs[0]);
    *pdf = pdfs[0] * pdfs[1];
}
```

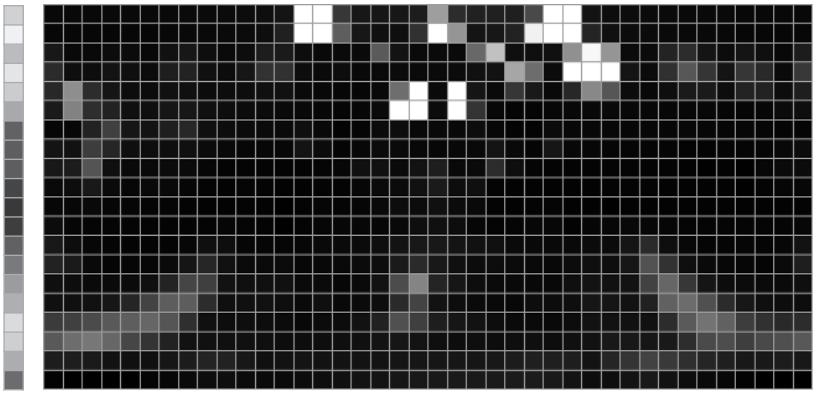
The value of the PDF for a given sample value is computed as the product of the conditional and marginal PDFs for sampling it from the distribution.

⟨*Distribution2D Public Methods*⟩ +≡ 672
```
float Pdf(float u, float v) const {
    int iu = Clamp(Float2Int(u * pConditionalV[0]->count), 0,
                   pConditionalV[0]->count-1);
    int iv = Clamp(Float2Int(v * pMarginal->count), 0,
                   pMarginal->count-1);
    if (pConditionalV[iv]->funcInt * pMarginal->funcInt == 0.f) return 0.f;
    return (pConditionalV[iv]->func[iu] * pMarginal->func[iv]) /
           (pConditionalV[iv]->funcInt * pMarginal->funcInt);
}
```

Clamp() 1000
Distribution1D 648
Distribution1D::count 648
Distribution1D::func 648
Distribution1D::funcInt 648
Distribution1D::
SampleContinuous() 649
Distribution2D::
pConditionalV 672
Distribution2D::
pMarginal 673
Float2Int() 1002



(a)



(b) (c)

Figure 13.16: The Piecewise-Constant Sampling Distribution for a High-Dynamic-Range Environment Map. (a) The original environment map, (b) a low-resolution version of the marginal density function $p[\hat{v}]$, (c) the conditional distributions for rows of the image. First the marginal 1D distribution (b) is used to select a v value, giving a row of the image to sample. Rows with bright pixels are more likely to be sampled. Then, given a row, a value u is sampled from that row's 1D distribution. *(Grace Cathedral environment map courtesy of Paul Debevec.)*

FURTHER READING

Many books have been written on Monte Carlo integration. Hammersley and Handscomb (1964), Spanier and Gelbard (1969), and Kalos and Whitlock (1986) are classic references. More recent books on the topic include those by Fishman (1996) and Liu (2001). Chib and Greenberg (1995) have written an approachable but rigorous introduction to the Metropolis algorithm. The Monte Carlo and Quasi Monte Carlo Web site is a useful gateway to recent work in the field *(www.mcqmc.org)*.

Good general references about Monte Carlo and its application to computer graphics are the theses by Lafortune (1996) and Veach (1997). This topic is also covered well by Dutré, Bekaert, and Bala (2006). Dutré's *Global Illumination Compendium* (2003) also has much

useful information related to this topic. The course notes from the Monte Carlo ray-tracing course at SIGGRAPH also have a wealth of practical information (Jensen et al. 2001a, 2003). The square to disk mapping was described by Shirley and Chiu (1997). Clarberg (2008) described a highly efficient implementation of a mapping from the square to the sphere that shares many of the important properties identified by Shirley and Chiu.

Steigleder and McCool (2003) described an alternative to the multidimensional sampling approach from Section 13.6.5: they linearized two and higher dimensional domains into 1D using a Hilbert curve and then sampled using 1D samples over the 1D domain. This leads to a simpler implementation that still maintains desirable stratification properties of the sampling distribution, thanks to the spatial coherence preserving properties of the Hilbert curve.

Lawrence et al. (2005) described an adaptive representation for CDFs, where the CDF is approximated with a piecewise linear function with fewer, but irregularly spaced, vertices compared to the complete CDF. This approach can substantially reduce storage requirements and improve lookup efficiency, taking advantage of the fact that large ranges of the CDF may be efficiently approximated with a single linear function.

Cline et al. (2009b) observed that the time spent in binary searches for sampling from precomputed distributions (like `Distribution1D` does) can take a substantial amount of execution time. (Indeed, `pbrt` spends approximately 18% of the time when rendering the TT car scene lit by an `InfiniteAreaLight` in the `Distribution1D::SampleContinuous()` method, which is used by `InfiniteAreaLight::Sample_L()`.) They presented two improved methods for doing this sort of search: the first stores a lookup table with n integer values, indexed by $\lfloor n\xi \rfloor$, which gives the first entry in the CDF array that is less than or equal to ξ. Starting a linear search from this offset in the CDF array can be much more efficient than a complete binary search over the entire array. They also presented a method based on approximating the inverse CDF as a piecewise linear function of ξ and thus enabling constant-time lookups at a cost of some accuracy (and thus some additional variance).

EXERCISES

Distribution1D 648

Distribution1D::
SampleContinuous() 649

InfiniteAreaLight 629

InfiniteAreaLight::
Sample_L() 727

● 2 13.1 Write a program that compares Monte Carlo and one or more alternative numerical integration techniques. Structure this program so that it is easy to replace the particular function being integrated. Verify that the different techniques compute the same result (given a sufficient number of samples for each of them). Modify your program so that it draws samples from distributions other than the uniform distribution for the Monte Carlo estimate and verify that it still computes the correct result when the correct estimator, Equation (13.3), is used. (Make sure that any alternative distributions you use have nonzero probability of choosing any value of x where $f(x) > 0$.)

● 1 13.2 Write a program that computes Monte Carlo estimates of the integral of a given function. Compute an estimate of the variance of the estimates by taking a

series of trials and using Equation (13.2) to compute variance. Demonstrate numerically that variance decreases at a rate of $O(\sqrt{n})$.

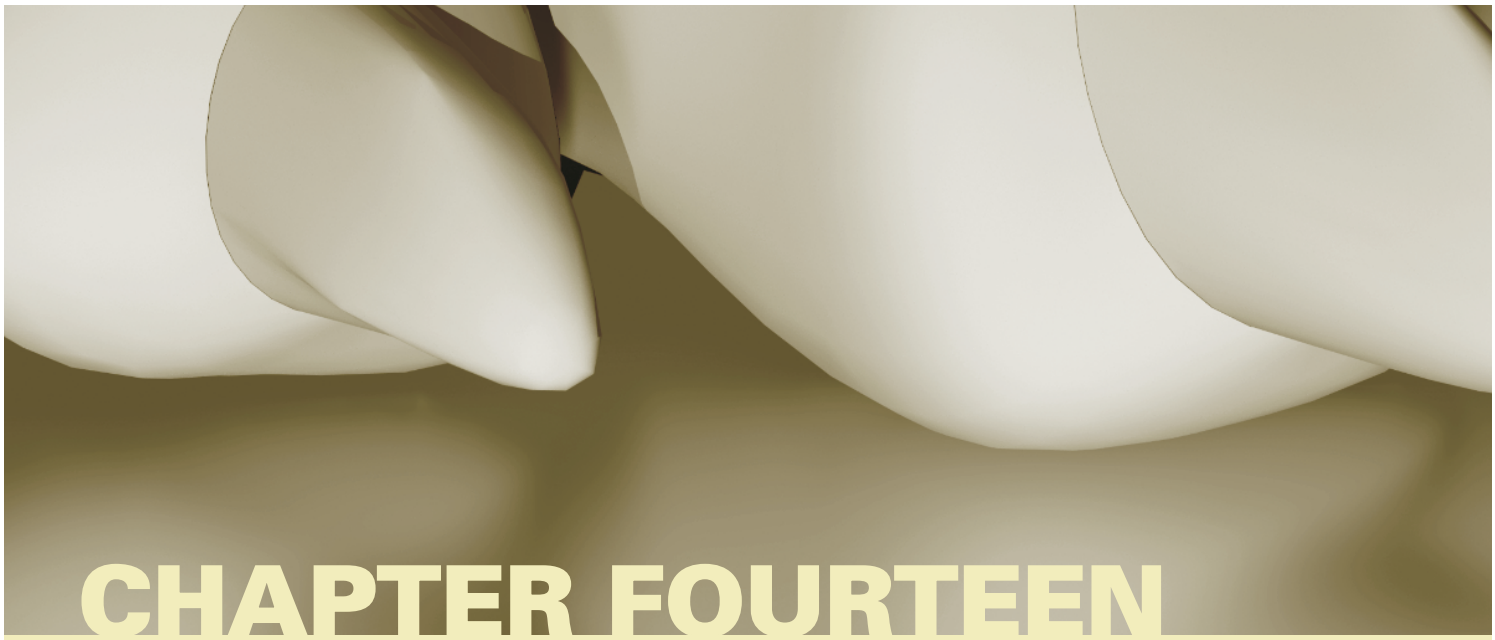
❶ 13.3 The depth-of-field code in Section 6.2.3 uses the `ConcentricSampleDisk()` function to generate samples on the circular lens, since this function gives less distortion than `UniformSampleDisk()`. Try replacing it with `UniformSampleDisk()` and measure the difference in image quality. For example, you might want to compare the error in images from using each approach and a relatively low number of samples to a highly sampled reference image.

Does `ConcentricSampleDisk()` in fact give less error in practice? Does it make a difference if a relatively simple scene is being rendered versus a very complex scene?

❷ 13.4 Modify the `Distribution1D` implementation to use the adaptive CDF representation described by Lawrence et al. (2005) and experiment with how much more compact the CDF representation can be made without causing image artifacts. (Good test scenes include those that use `InfiniteAreaLights`, which use the `Distribution2D` and, thus, `Distribution1D` for sampling.) Can you measure an improvement in rendering speed due to more efficient searches through the approximated CDF?

ConcentricSampleDisk() 667
Distribution1D 648
Distribution2D 672
InfiniteAreaLight 629
UniformSampleDisk() 666

This page intentionally left blank



CHAPTER FOURTEEN

14 MONTE CARLO INTEGRATION II: IMPROVING EFFICIENCY

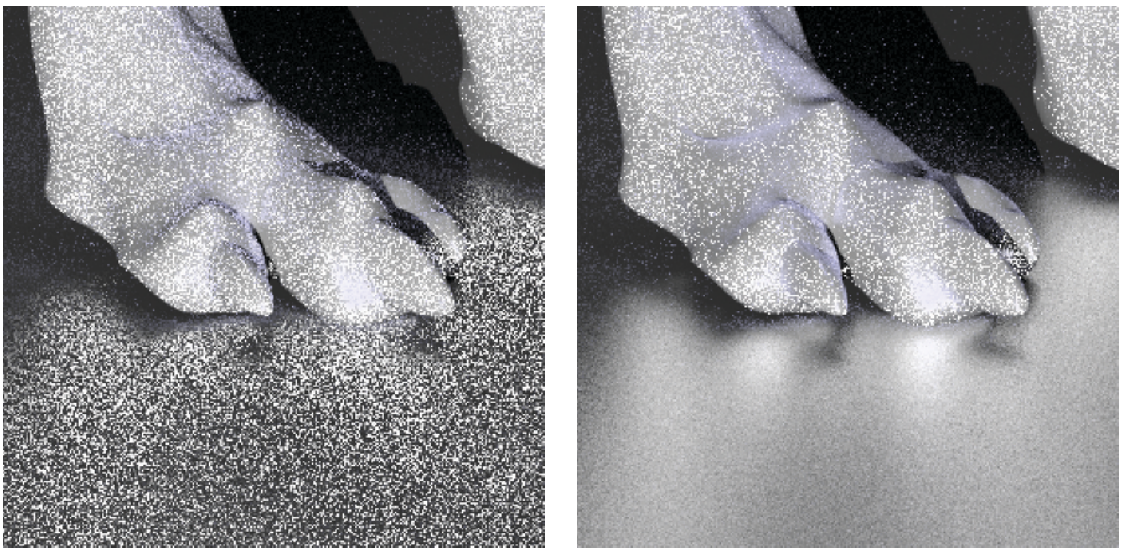
Variance in Monte Carlo ray tracing manifests itself as noise in the image—an unpleasant artifact (Figure 14.1). The battle against variance is the basis of most of the work in optimizing Monte Carlo. Recall that Monte Carlo's convergence rate means that it is necessary to quadruple the number of samples in order to reduce the variance by half. Because the run time of the estimation procedure is proportional to the number of samples, the cost of reducing variance can be high. This chapter will develop the theory and practice of techniques for improving the efficiency of Monte Carlo integration without necessarily increasing the number of samples.

The *efficiency* of an estimator F is defined as

$$\epsilon[F] = \frac{1}{V[F]T[F]},$$

where $V[F]$ is its variance and $T[F]$ is the running time to compute its value. According to this metric, an estimator F_1 is more efficient than F_2 if it takes less time to produce the same variance, or if it produces less variance in the same amount of time.

One of the techniques that has been most effective for improving efficiency for rendering problems is a method called *importance sampling*. Recall that there is some freedom in the choice of distribution $p(x)$ to use as the sample distribution for the Monte Carlo estimator in Equation (13.3). It turns out that choosing a sampling distribution that is similar in shape to the integrand leads to reduced variance. This technique is called importance sampling because samples tend to be taken in "important" parts of the function's domain, where the function's value is relatively large.



(a) (b)

Figure 14.1: (a) A scene with glossy reflection, where variance from estimating the value of the scattering equation at each pixel is visually disturbing. (b) An image computed with the variance reduction techniques described in this chapter; it has substantially less variance but uses the same amount of computation.

The first half of this chapter discusses importance sampling and a number of other techniques for improving the efficiency of Monte Carlo. The second half of the chapter then derives techniques for generating samples according to the distributions of BSDFs, light sources, and functions related to volume scattering so that they can be used as sampling distributions for importance sampling. These sampling methods will be used throughout Chapters 15 through 17.

14.1 RUSSIAN ROULETTE AND SPLITTING

Russian roulette and splitting are two related techniques that can improve the efficiency of Monte Carlo estimates by increasing the likelihood that each sample will have a significant contribution to the result. Russian roulette addresses the problem of samples that are expensive to evaluate but make a small contribution to the final result. We would like to avoid the work of evaluating these samples while still computing a correct estimator.

As an example, consider the problem of estimating the direct lighting integral, which gives reflected radiance at a point due to direct lighting from the light sources in the scene, L_{d}:

$$L_{\mathrm{o}}(\mathrm{p}, \omega_{\mathrm{o}}) = \int_{S^2} f_{\mathrm{r}}(\mathrm{p}, \omega_{\mathrm{o}}, \omega_{\mathrm{i}})\, L_{\mathrm{d}}(\mathrm{p}, \omega_{\mathrm{i}})\, |\cos\theta_i|\, \mathrm{d}\omega_{\mathrm{i}}.$$

Assume that we have decided to take $N = 2$ samples from some distribution $p(\omega)$ to compute the estimator

$$\frac{1}{2}\sum_{i=1}^{2}\frac{f_{\mathrm{r}}(\mathrm{p},\omega_{\mathrm{o}},\omega_i)\,L_{\mathrm{d}}(\mathrm{p},\omega_i)\,|\cos\theta_i|}{p(\omega_i)}.$$

Most of the computation expense of each of the terms of the sum comes from tracing a shadow ray from the point p to see if the light source is occluded as seen from p.

For all of the directions ω_i where the integrand's value is clearly zero (e.g., because $f_{\mathrm{r}}(\mathrm{p},\omega_{\mathrm{o}},\omega_i)$ is zero for that direction), we can skip the work of tracing the shadow ray, since tracing it won't change the final value computed. Russian roulette makes it possible to also skip tracing rays when the integrand's value is very low but not necessarily zero, while still computing the correct value on average. For example, we might want to avoid tracing rays where $f_{\mathrm{r}}(\mathrm{p},\omega_{\mathrm{o}},\omega_i)$ is small, or when ω_i is close to the horizon and thus $|\cos\theta_i|$ is small. Of course, these samples just can't be ignored completely, since the estimator would then consistently underestimate the correct result.

To apply Russian roulette, we select some termination probability q. This value can be chosen in almost any manner; for example, it could be based on an estimate of the value of the integrand for the particular sample chosen, increasing as the integrand's value becomes smaller. With probability q, the integrand is not evaluated for the particular sample and some constant value c is used in its place ($c = 0$ is often used). With probability $1-q$, the integrand is still evaluated but is weighted by a term, $1/(1-q)$, that effectively accounts for all of the samples that were skipped:

$$F' = \begin{cases} \frac{F-qc}{1-q} & \xi > q \\ c & \text{otherwise.} \end{cases}$$

The expected value of the resulting estimator is the same as the expected value of the original estimator:

$$E[F'] = (1-q)\left(\frac{E[F]-qc}{1-q}\right) + qc = E[F].$$

Russian roulette never reduces variance. In fact, unless somehow $c = F$, it will always increase variance. However, it can improve efficiency if probabilities are chosen so that samples that are likely to make a small contribution to the final result are skipped.

One pitfall is that poorly chosen Russian roulette weights can substantially increase variance. Imagine applying Russian roulette to all of the camera rays with a termination probability of .99: we'd only trace 1% of the camera rays, weighting each of them by $1/.01 = 100$. The resulting image would still be "correct" in a strictly mathematical sense, although visually the result would be terrible: mostly black pixels with a few very bright ones. One of the exercises at the end of the next chapter discusses this problem further and describes a technique called *efficiency-optimized Russian roulette* that tries to set Russian roulette weights in a way that minimizes the increase in variance.

14.1.1 SPLITTING

While Russian roulette reduces the effort spent evaluating unimportant samples, splitting increases the number of samples taken in order to improve efficiency. Consider again the problem of computing reflection due only to direct illumination. Ignoring pixel filtering, this problem can be written as a double integral over the area of the pixel A and over

the sphere of directions $\mathcal{S}^2$ at the visible points on surfaces at each (x, y) pixel position, where $L_{\mathrm{d}}(x, y, \omega)$ denotes the exitant radiance at the object visible at the position (x, y) on the image due to incident radiance from the direction ω:

$$\int_A \int_{\mathcal{S}^2} L_{\mathrm{d}}(x, y, \omega) \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}\omega.$$

The natural way to estimate the integral is to generate N samples and apply the Monte Carlo estimator, where each sample consists of an (x, y) image position and a direction ω toward a light source. If there are many light sources in the scene, or if there is an area light casting soft shadows, tens or hundreds of samples may be needed to compute a result with an acceptable variance level. Unfortunately, each sample requires that two rays be traced through the scene: one to compute the first visible surface from position (x, y) on the image plane, and one a shadow ray along ω to a light source.

The problem with this approach is that if $N = 100$ samples are taken to estimate this integral, then 200 rays will be traced: 100 camera rays and 100 shadow rays. Yet, 100 camera rays may be many more than are needed for good pixel antialiasing and thus may make relatively little contribution to variance reduction in the final result. Splitting addresses this problem by formalizing the approach of taking multiple samples in some of the dimensions of integration for each sample taken in other dimensions.

With splitting, the estimator for this integral can be written taking N image samples and M light samples per image sample:

$$\frac{1}{N} \frac{1}{M} \sum_{i=1}^{N} \sum_{j=1}^{M} \frac{L(x_i, y_i, \omega_{i,j})}{p(x_i, y_i) \, p(\omega_{i,j})}.$$

Thus, we could take just 5 image samples, but take 20 light samples per image sample, for a total of 105 rays traced, rather than 200, while still taking 100 area light samples in order to compute a high-quality soft shadow.

14.2 CAREFUL SAMPLE PLACEMENT

A classic and effective family of techniques for variance reduction is based on the careful placement of samples in order to better capture "important" features of the integrand (or, more accurately, to be less likely to miss important features). These techniques are complementary to techniques like importance sampling and are used extensively in `pbrt`. Indeed, one of the tasks of `Samplers` in Chapter 7 was to generate well-distributed samples for use by the integrators for just this reason, although at the time we offered only an intuitive sense of why this was worthwhile. Here we will formally justify that extra work.

14.2.1 STRATIFIED SAMPLING

Stratified sampling was first introduced in Section 7.3, and we now have the tools to motivate its use. Stratified sampling works by subdividing the integration domain Λ into n nonoverlapping regions $\Lambda_1, \Lambda_2, \ldots, \Lambda_n$. Each region is called a *stratum*, and they

Sampler 340

must completely cover the original domain:

$$\bigcup_{i=1}^{n} \Lambda_i = \Lambda.$$

To draw samples from Λ, we will draw n_i samples from each Λ_i, according to densities p_i inside each stratum. A simple example is supersampling a pixel. With stratified sampling, the area around a pixel is divided into a $k \times k$ grid, and a sample is drawn from each grid cell. This is better than taking k^2 random samples, since the sample locations are less likely to clump together. Here we will show why this technique reduces variance.

Within a single stratum Λ_i, the Monte Carlo estimate is

$$F_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \frac{f(X_{i,j})}{P_i(X_{i,j})},$$

where $X_{i,j}$ is the jth sample drawn from density p_i. The overall estimate is $F = \sum_{i=1}^{n} v_i F_i$, where v_i is the fractional volume of stratum i ($v_i \in (0, 1]$).

The true value of the integrand in stratum i is

$$\mu_i = E\left[f\left(X_{i,j}\right)\right] = \frac{1}{v_i} \int_{\Lambda_i} f(x)\, \mathrm{d}x,$$

and the variance in this stratum is

$$\sigma_i^2 = \frac{1}{v_1} \int_{\Lambda_i} \left(f(x) - \mu_i\right)^2 \mathrm{d}x.$$

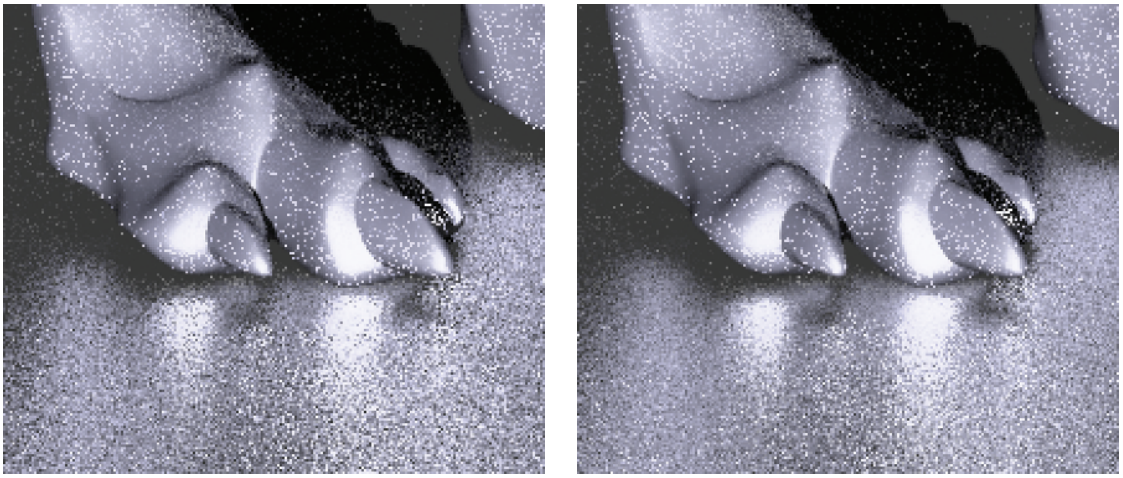
Thus, with n_i samples in the stratum, the variance of the per-stratum estimator is σ_i^2/n_i. This shows that the variance of the overall estimator is

$$\begin{aligned} V[F] &= V\left[\sum v_i F_i\right] \\ &= \sum V\left[v_i F_i\right] \\ &= \sum v_i^2 V\left[F_i\right] \\ &= \sum \frac{v_i^2 \sigma_i^2}{n_i}. \end{aligned}$$

If we make the reasonable assumption that the number of samples n_i is proportional to the volume v_i, then we have $n_i = v_i N$, and the variance of the overall estimator is

$$V\left[F_N\right] = \frac{1}{N} \sum v_i \sigma_i^2.$$

To compare this result to the variance without stratification, we note that choosing an unstratified sample is equivalent to choosing a random stratum I according to the discrete probability distribution defined by the volumes v_i, and then choosing a random sample X in Λ_I. In this sense, X is chosen *conditionally* on I, so it can be shown using



(a) (b)

Figure 14.2: Variance is higher and the image noisier (a) when random sampling is used to compute the effect of glossy reflection than (b) when a stratified distribution of sample directions is used instead. (Compare the edges of the highlights on the ground, for example.)

conditional probability that

$$\begin{aligned} V[F] &= E_x V_i F + V_x E_i F \\ &= \frac{1}{N}\left[\sum v_i \sigma_i^2 + \sum v_i \left(\mu_i - Q\right)\right], \end{aligned}$$

where Q is the mean of f over the whole domain Λ. See Veach (1997) for a derivation of this result.

There are two things to notice about this expression. First, we know that the right-hand sum must be nonnegative, since variance is always nonnegative. Second, it demonstrates that stratified sampling can never increase variance. In fact, stratification always reduces variance unless the right-hand sum is exactly zero. It can only be zero when the function f has the same mean over each stratum Λ_i. In fact, for stratified sampling to work best, we would like to maximize the right-hand sum, so it is best to make the strata have means that are as unequal as possible. This explains why *compact* strata are desirable if one does not know anything about the function f. If the strata are wide, they will contain more variation and will have μ_i closer to the true mean Q.

Figure 14.2 shows the effect of using stratified sampling versus a uniform random distribution for sampling ray directions for glossy reflection. There is a reasonable reduction in variance at essentially no cost in running time.

The main downside of stratified sampling is that it suffers from the same "curse of dimensionality" as standard numeric quadrature. Full stratification in D dimensions with S strata per dimension requires S^D samples, which quickly becomes prohibitive. Fortunately, it is often possible to stratify some of the dimensions independently and then randomly associate samples from different dimensions, as was done in Section 7.3.

Choosing which dimensions are stratified should be done in a way that stratifies dimensions that tend to be most highly correlated in their effect on the value of the integrand (Owen 1998). For example, for the direct lighting example in Section 14.1.1, it is far more effective to stratify the (x, y) pixel positions and to stratify the (θ, ϕ) ray direction—stratifying (x, θ) and (y, ϕ) would almost certainly be ineffective.

Another solution to the curse of dimensionality that has many of the same advantages of stratification is to use Latin hypercube sampling (also introduced in Section 7.3), which can generate any number of samples independent of the number of dimensions. Unfortunately, Latin hypercube sampling isn't as effective as stratified sampling at reducing variance, especially as the number of samples taken becomes large. Nevertheless, Latin hypercube sampling is provably no worse than uniform random sampling and is often much better.

14.2.2 QUASI MONTE CARLO

The low-discrepancy sampling techniques introduced in Section 7.4 are the foundation of a branch of Monte Carlo called *quasi Monte Carlo*. The key component of quasi–Monte Carlo techniques is that they replace the random numbers used in standard Monte Carlo with low-discrepancy point sets generated by carefully designed deterministic algorithms.

The advantage of this approach is that for many integration problems, quasi-Monte Carlo techniques have asymptotically faster rates of convergence than methods based on standard Monte Carlo. Many of the techniques used in regular Monte Carlo algorithms can be shown to work equally well with quasi-random sample points, including importance sampling. Some others (e.g., Russian roulette and rejection sampling) cannot. While the asymptotic convergence rates are not generally applicable to the discontinuous integrands in graphics because the convergence rates depend on smoothness properties in the integrand, quasi Monte Carlo generally performs better than regular Monte Carlo for these integrals in practice. The "Further Reading" section at the end of this chapter has more information about this topic.

In `pbrt`, we have generally glossed over the differences between these two approaches and have localized them in the `Sampler`s in Chapter 7. This introduces the possibility of subtle errors if a `Sampler` generates quasi-random sample points that an `Integrator` then improperly uses as part of an implementation of an algorithm that is not suitable for quasi Monte Carlo. As long as `Integrator`s only use these sample points for importance sampling or other techniques that are applicable in both approaches, this isn't a problem.

14.2.3 WARPING SAMPLES AND DISTORTION

When applying stratified sampling or low-discrepancy sampling to problems like choosing points on light sources for integration for area lighting, `pbrt` generates a set of samples (ξ_1, ξ_2) over the domain $[0, 1]^2$ and then uses algorithms based on the transformation methods introduced in Sections 13.5 and 13.6 to map these samples to points on the light source. Implicit in this process is the expectation that the transformation to points on the light source will generally preserve the stratification properties of the samples from $[0, 1]^2$—in other words, nearby samples should map to nearby positions on the surface

Integrator 740
Sampler 340

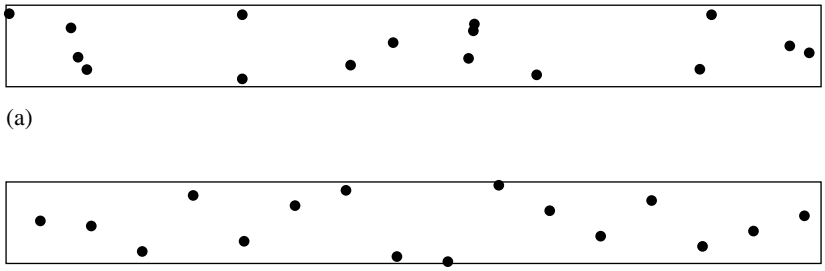


Figure 14.3: (a) Transforming a 4×4 stratified sampling pattern to points on a long and thin quadrilateral light source effectively gives less than 16 well-distributed samples; stratification in the vertical direction is not helpful. (b) Samples from a $(0, 2)$-sequence remain well distributed even after this transformation.

of the light, and faraway samples should map to far-apart positions on the light. If the mapping does not preserve this property, then the benefits of stratification are lost.

For example, this explains why Shirley's square-to-circle mapping (Figure 13.13) is better than the straightforward mapping (Figure 13.12), since the straightforward mapping has less compact strata away from the center. It also explains why $(0, 2)$-sequences can be so much better than stratified patterns in practice: they are more robust with respect to preserving their good distribution properties after being transformed to other domains. For example, Figure 14.3 shows what happens when a set of 16 well-distributed sample points are transformed to be points on a skinny quadrilateral by scaling them to cover its surface; the $(0, 2)$-sequence remains well-distributed, but the stratified pattern fares much less well.

14.3 BIAS

Another approach to variance reduction is to introduce *bias* into the computation: sometimes knowingly computing an estimate that doesn't actually have an expected value equal to the desired quantity can nonetheless lead to lower variance. An estimator is *unbiased* if its expected value is equal to the correct answer. If not, the difference

$$\beta = E[F] - \int f(x)\, \mathrm{d}x$$

is the amount of bias.

Kalos and Whitlock (1986, pp. 36–37) gave the following example of how bias can sometimes be desirable. Consider the problem of computing an estimate of the mean value of a distribution $X_i \sim p$ over the interval from zero to one. One could use the estimator

$$\frac{1}{N} \sum_{i=1}^{N} X_i,$$

or one could use the biased estimator

$$\frac{1}{2}\max(X_1, X_2, \ldots, X_N).$$

The first estimator is in fact unbiased, but has variance with order $O(N^{-1})$. The second estimator's expected value is

$$0.5\frac{N}{N+1} \neq 0.5,$$

so it is biased, although its variance is $O(N^{-2})$, which is much better. For large values of N, the second estimator may be preferred.

The pixel reconstruction method described in Section 7.7 can also be thought of as a biased estimator. Considering it as a Monte Carlo estimation problem, we'd like to compute an estimate of

$$I(x, y) = \iint f(x - x', y - y')L(x', y')\,\mathrm{d}x'\,\mathrm{d}y',$$

where $I(x, y)$ is a final pixel value, $f(x, y)$ is the pixel filter function (which we assume here to be normalized to integrate to one), and $L(x, y)$ is the image radiance function.

Assuming we have chosen image plane samples uniformly, all samples have the same probability density, which we will denote by p_c. Thus, the unbiased Monte Carlo estimator of this equation is

$$I(x, y) \approx \frac{1}{Np_c}\sum_{i=1}^{N} f(x - x_i, y - y_i)L(x_i, y_i).$$

This gives a different result than the pixel filtering equation we used previously, Equation (7.4), which was

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i)L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}.$$

Yet, the biased estimator is preferable in practice because it gives a result with less variance. For example, if all radiance values $L(x_i, y_i)$ have a value of one, the biased estimator will always reconstruct an image where all pixel values are exactly one—clearly a desirable property. However, the unbiased estimator will reconstruct pixel values that are not all one, since the sum

$$\sum_i f(x - x_i, y - y_i)$$

will generally not be equal to p_c and thus will have a different value due to variation in the filter function depending on the particular (x_i, y_i) sample positions used for the pixel. Thus, the variance due to this effect leads to an undesirable result in the final image. Even for more complex images, the variance that would be introduced by the unbiased estimator is a more objectionable artifact than the bias from Equation (7.4).

14.4 IMPORTANCE SAMPLING

Importance sampling is a variance reduction technique that exploits the fact that the Monte Carlo estimator

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

converges more quickly if the samples are taken from a distribution $p(x)$ that is similar to the function $f(x)$ in the integrand. The basic idea is that by concentrating work where the value of the integrand is relatively high, an accurate estimate is computed more efficiently (Figure 14.4).

For example, suppose we are evaluating the scattering equation, Equation (5.8). Consider what happens when this integral is estimated; if a direction is randomly sampled that is nearly perpendicular to the surface normal, the cosine term will be close to zero. All the expense of evaluating the BSDF and tracing a ray to find the incoming radiance at that sample location will be essentially wasted, as the contribution to the final result will be minuscule. As such, we would be better served if we sampled the sphere in a way that reduced the likelihood of choosing directions near the equator. More generally, if directions are sampled from distributions that match other factors of the integrand (the BSDF, the incoming illumination distribution, etc.), efficiency is similarly improved.

So long as the random variables are sampled from a probability distribution that is similar in shape to the integrand, variance is reduced. We will not provide a rigorous proof of this fact but will instead present an informal and intuitive argument. Suppose we're trying to use Monte Carlo techniques to evaluate some integral $\int f(x)\,\mathrm{d}x$. Since we have freedom in choosing a sampling distribution, consider the effect of using a distribution $p(x) \propto f(x)$, or $p(x) = cf(x)$. It is trivial to show that normalization forces

$$c = \frac{1}{\int f(x)\,\mathrm{d}x}.$$

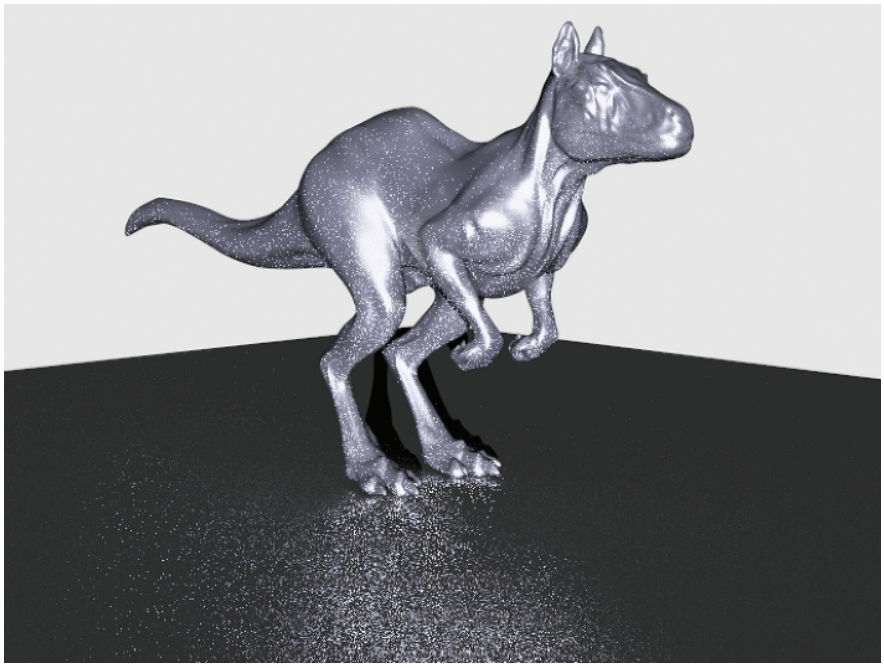
Finding such a PDF requires that we know the value of the integral, which is what we were trying to estimate in the first place. Nonetheless, for the purposes of this example, if we *could* sample from this distribution, each estimate would have the value

$$\frac{f(X_i)}{p(X_i)} = \frac{1}{c} = \int f(x)\,\mathrm{d}x.$$

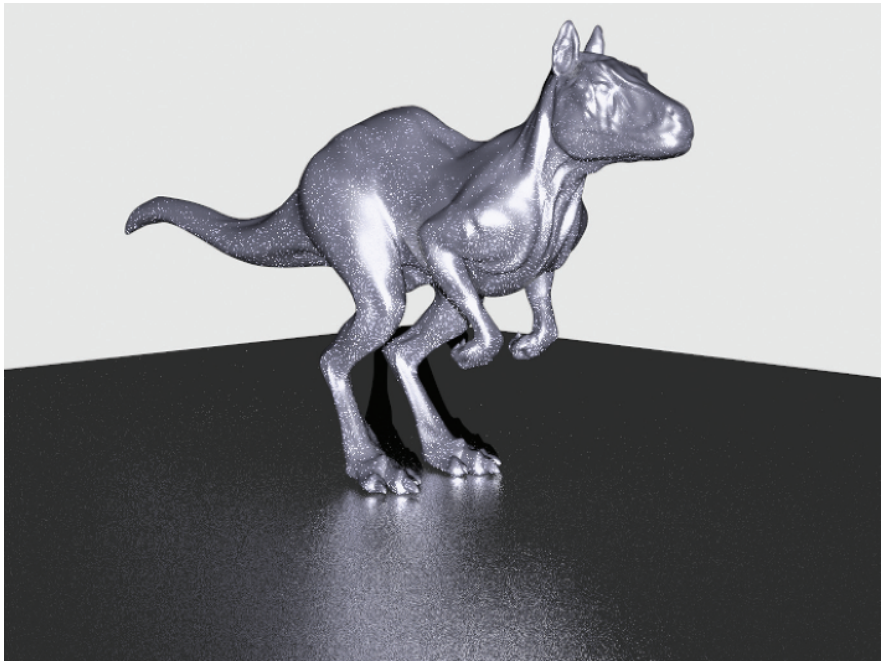
Since c is a constant, each estimate has the same value, and the variance is zero! Of course, this is ludicrous since we wouldn't bother using Monte Carlo if we could integrate f directly. However, if a density $p(x)$ can be found that is similar in shape to $f(x)$, variance decreases.

Importance sampling can increase variance if a poorly chosen distribution is used. To understand the effect of sampling from a PDF that is a poor match for the integrand, consider using the distribution

$$p(x) = \begin{cases} 99.01 & x \in [0, .01) \\ .01 & x \in [.01, 1) \end{cases}$$



(a)



(b)

Figure 14.4: (a) Using a stratified uniform distribution of rays over the hemisphere gives an image with much more variance than (b) applying importance sampling and choosing stratified rays from a distribution based on the BRDF.

to compute the estimate of $\int f(x)\,\mathrm{d}x$ where

$$f(x) = \begin{cases} .01 & x \in [0, .01) \\ 1.01 & x \in [.01, 1). \end{cases}$$

By construction, the value of the integral is one, yet using $p(x)$ to draw samples to compute a Monte Carlo estimate will give a terrible result: almost all of the samples will be in the range [0, .01), where the estimator has the value $f(x)/p(x) \approx 0.0001$. For any estimate where none of the samples ends up being outside of this range, the result will be very inaccurate, almost 10,000 times smaller than it should be. Even worse is the case where some samples do end up being taken in the range [.01, 1). This will happen rarely, but when it does, we have the combination of a relatively high value of the integrand and a relatively low value of the PDF, $f(x)/p(x) = 101$. A large number of samples would be necessary to balance out these extremes to reduce variance enough to get a result close to the actual value, one.

Fortunately, it's not too hard to find good sampling distributions for importance sampling for many integration problems in graphics. For example, in many cases, the integrand is the product of more than one function. It can be difficult to construct a PDF that is similar to the complete product, but finding one that is similar to one of the multiplicands can be helpful. This will be a common strategy in the light transport algorithms in the following chapters, where we will be trying to estimate integrals that multiply lighting, visibility, scattering, and cosine terms. (The "Further Reading" section at the end of this chapter has pointers to methods that generate PDFs that sample products of functions like these.)

In practice, importance sampling is one of the most frequently used variance reduction techniques in rendering, since it is easy to apply and is very effective when good sampling distributions are used. It is one of the variance reduction techniques of choice in `pbrt`, and therefore a variety of techniques for sampling from distributions from BSDFs, light sources, and functions related to participating media will be derived in this chapter.

14.4.1 MULTIPLE IMPORTANCE SAMPLING

Monte Carlo provides tools to estimate integrals of the form $\int f(x)\,\mathrm{d}x$. However, we are frequently faced with integrals that are the product of two or more functions: $\int f(x)g(x)\,\mathrm{d}x$. If we have an importance sampling strategy for $f(x)$ *and* a strategy for $g(x)$, which should we use? (Assume that we are not able to combine the two sampling strategies to compute a PDF that is proportional to the product $f(x)g(x)$ that can itself be sampled easily.) As shown in the discussion of importance sampling, a bad choice of sampling distribution can be much worse than just using a uniform distribution.

For example, consider the problem of evaluating direct lighting integrals of the form

$$L_\mathrm{o}(\mathrm{p}, \omega_\mathrm{o}) = \int_{\mathbb{S}^2} f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{d}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}.$$

If we were to perform importance sampling to estimate this integral according to distributions based on either L_d or f_r, one of these two will often perform poorly.

Consider a near-mirror BRDF illuminated by an area light where L_{d}'s distribution is used to draw samples. Because the BRDF is almost a mirror, the value of the integrand will be close to zero at all ω_{i} directions except those around the perfect specular reflection direction. This means that almost all of the directions sampled by L_{d} will have zero contribution, and variance will be quite high. Even worse, as the light source grows large and a larger set of directions is potentially sampled, the value of the PDF decreases, so for the rare directions where the BRDF is nonzero for the sampled direction we will have a large integrand being divided by a small PDF value. While sampling from the BRDF's distribution would be a much better approach to this particular case, for diffuse or glossy BRDFs and small light sources, sampling from the BRDF's distribution can similarly lead to much higher variance than sampling from the light's distribution.

Unfortunately, the obvious solution of taking some samples from each distribution and averaging the two estimators is hardly any better. Because the variance is additive in this case, this approach doesn't help—once variance has crept into an estimator, we can't eliminate it by adding it to another estimator even if it itself has low variance.

Multiple importance sampling (MIS) addresses exactly these kinds of problems, with a simple and easy-to-implement technique. The basic idea is that, when estimating an integral, we should draw samples from multiple sampling distributions, chosen in the hope that at least one of them will match the shape of the integrand reasonably well, even if we don't know which one this will be. MIS provides a method to weight the samples from each technique that can eliminate large variance spikes due to mismatches between the integrand's value and the sampling density. Specialized sampling routines that only account for unusual special cases are even encouraged, as they reduce variance when those cases occur, with relatively little cost in general. Figure 14.5 shows the reduction in variance from using MIS to compute reflection from direct illumination compared to the two approaches described previously.

If two sampling distributions p_f and p_g are used to estimate the value of $\int f(x)g(x)\,\mathrm{d}x$, the new Monte Carlo estimator given by MIS is

$$\frac{1}{n_f}\sum_{i=1}^{n_f}\frac{f(X_i)g(X_i)w_f(X_i)}{p_f(X_i)}+\frac{1}{n_g}\sum_{j=1}^{n_g}\frac{f(Y_j)g(Y_j)w_g(Y_j)}{p_g(Y_j)},$$

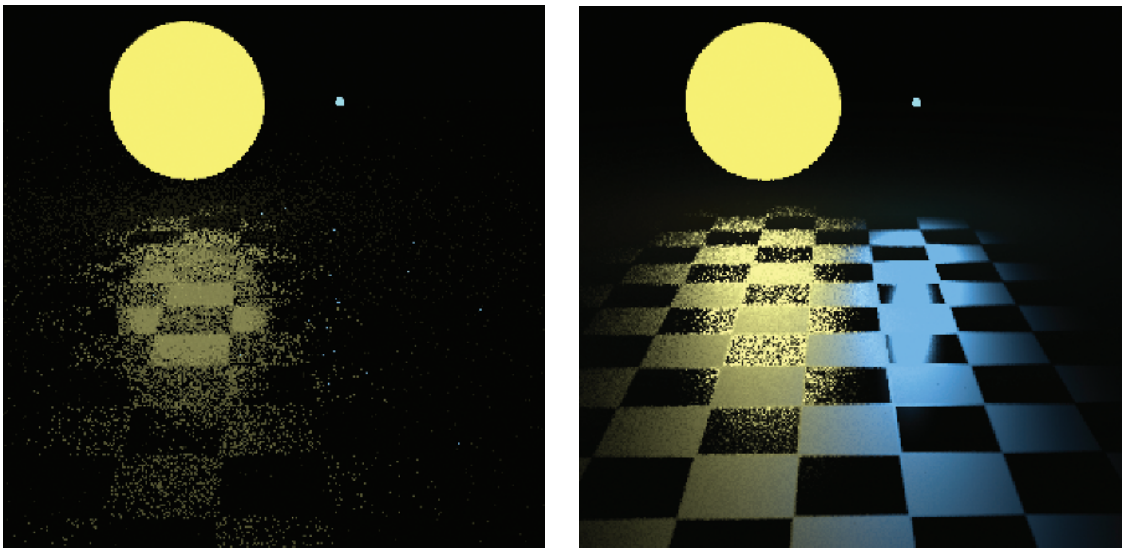
where n_f is the number of samples taken from the p_f distribution method, n_g is the number of samples taken from p_g, and w_f and w_g are special weighting functions chosen such that the expected value of this estimator is the value of the integral of $f(x)g(x)$.

The weighting functions take into account *all* of the different ways that a sample X_i or Y_j could have been generated, rather than just the particular one that was actually used. A good choice for this weighting function is the *balance heuristic*:

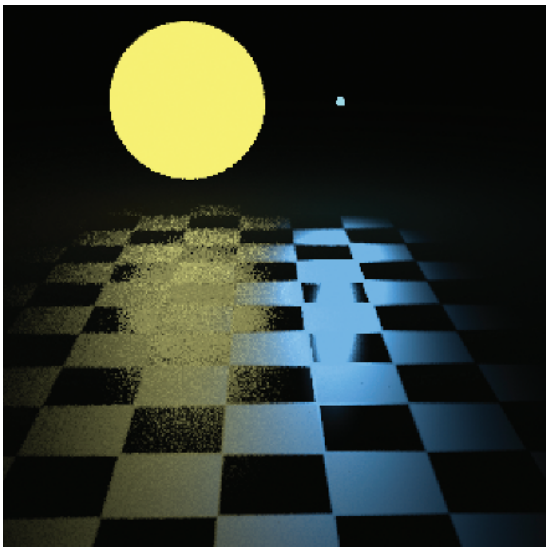
$$w_s(x)=\frac{n_s p_s(x)}{\sum_i n_i p_i(x)}.$$

The balance heuristic is a provably good way to weight samples to reduce variance.

Consider the effect of this term for the case where a sample X has been drawn from the p_f distribution at a point where the value $p_f(X)$ is relatively low. Assuming that p_f is



(a) (b)



(c)

Figure 14.5: Multiple importance sampling is a very effective modification to standard importance sampling methods: (a) A scene with a glossy checkerboard where some checks are nearly perfectly specular and others are nearly diffuse, where direct lighting is computed by sampling the BSDF. This strategy works well for the nearly specular checks but not for the diffuse ones. (b) The result of sampling from the light sources' distributions instead. Again, some reflections look good while others suffer from very high variance. (c) The result of using multiple importance sampling with the power heuristic, giving the best overall result.

a good match for the shape of $f(x)$, then the value of $f(X)$ will also be relatively low. But suppose that $g(X)$ has a relatively high value. The standard importance sampling estimate

$$\frac{f(X)g(X)}{p_f(X)}$$

will have a very large value due to $p_f(X)$ being small, and we will have high variance.

With the balance heuristic, the contribution of X will be

$$\frac{f(X)g(X)w_f(X)}{p_f(X)} = \frac{f(X)g(X)\, n_f\, p_f(X)}{p_f(X)(n_f\, p_f(X) + n_g\, p_g(X))} = \frac{f(X)g(X)\, n_f}{n_f\, p_f(X) + n_g\, p_g(X)}.$$

As long as p_g's distribution is a reasonable match for $g(x)$, then the denominator won't be too small thanks to the $n_g p_g(X)$ term, and the huge variance spike is eliminated, even though X was sampled from a distribution that was in fact a poor match for the integrand. The fact that another distribution will also be used to generate samples and that this new distribution will likely find a large value of the integrand at X are brought together in the weighting term to reduce the variance problem.

Here we provide an implementation of the balance heuristic for the specific case of two distributions p_f and p_g. We will not need a more general multidistribution case in `pbrt`.

⟨*Monte Carlo Inline Functions*⟩ ≡
```
inline float BalanceHeuristic(int nf, float fPdf, int ng, float gPdf) {
    return (nf * fPdf) / (nf * fPdf + ng * gPdf);
}
```

In practice, the *power heuristic* often reduces variance even further. For an exponent β, the power heuristic is

$$w_s(x) = \frac{(n_s p_s(x))^\beta}{\sum_i (n_i p_i(x))^\beta}.$$

Veach determined empirically that $\beta = 2$ is a good value. We have $\beta = 2$ hard-coded into the implementation here.

⟨*Monte Carlo Inline Functions*⟩ +≡
```
inline float PowerHeuristic(int nf, float fPdf, int ng, float gPdf) {
    float f = nf * fPdf, g = ng * gPdf;
    return (f*f) / (f*f + g*g);
}
```

14.5 SAMPLING REFLECTION FUNCTIONS

Because importance sampling is such an effective variance reduction technique, it's worth going through the effort to find sampling distributions that match functions in the common integrands in graphics. In this section, we will show the derivations and implementations for sampling strategies corresponding to distributions that are similar to many of the BSDF models used in `pbrt`.

The BxDF::Sample_f() method randomly chooses a direction according to a distribution that is similar to its corresponding scattering function. In Section 8.2, this method was used for finding reflected and transmitted rays from perfectly specular surfaces; later in this section, we will show how that sampling process is a special case of the sampling techniques of the previous chapter. BxDF::Sample_f() takes two sample values in the range $[0, 1)^2$ that are intended to be used by a transformation-based sampling algorithm. Since these are parameters to the method, the routine calling it can easily use stratified or low-discrepancy sampling techniques to generate them, thus ensuring well-distributed directions. If the algorithm used by BxDF::Sample_f() to generate directions doesn't need these sample values (e.g., if it uses rejection sampling), it can just ignore these arguments, although stratified sampling of a distribution that only roughly matches the BSDF is generally a more effective approach than rejection sampling from its distribution exactly.

This method returns the sampled direction in *wi and returns the value of $p(\omega_i)$ in *pdf. The value of the BSDF for the chosen direction is returned with a Spectrum return value. The PDF value returned should be measured with respect to solid angle on the hemisphere, and both the outgoing direction ω_o and the sampled incident direction ω_i should be in the standard reflection coordinate system (see "Geometric Setting," page 425).

The default implementation of this method samples the unit hemisphere with a cosine-weighted distribution. Samples from this distribution will give correct results for any BRDF that isn't described by a delta distribution, since there is some probability of sampling all directions where the BRDF's value is nonzero: $p(\omega) > 0$ for all ω. (BTDFs will thus always need to override this method but can sample the opposite hemisphere uniformly if they don't have a better sampling method.)

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum BxDF::Sample_f(const Vector &wo, Vector *wi,
                        float u1, float u2, float *pdf) const {
    ⟨Cosine-sample the hemisphere, flipping the direction if necessary 694⟩
    *pdf = Pdf(wo, *wi);
    return f(wo, *wi);
}
```

There is a subtlety related to the orientation of the normal in the reflection coordinate system that must be accounted for here: the direction returned by Malley's method will always be in the hemisphere around (0, 0, 1). If the ω_o direction is in the opposite hemisphere, then ω_i must be flipped to lie in the same hemisphere as ω_o. This issue is a direct consequence of the fact that pbrt does not flip the normal to be on the same side of the surface as the ω_o direction.

⟨*Cosine-sample the hemisphere, flipping the direction if necessary*⟩ ≡ 694, 701
```
*wi = CosineSampleHemisphere(u1, u2);
if (wo.z < 0.) wi->z *= -1.f;
```

While BxDF::Sample_f() returns the value of the PDF for the direction it chose, the BxDF::Pdf() method returns the value of the PDF for an arbitrary given direction. This method is useful for multiple importance sampling, where it is necessary to be able to find the PDF for directions sampled from other distributions. It is crucial that any

BxDF 428
BxDF::f() 429
BxDF::Pdf() 695
BxDF::Sample_f() 694
Spectrum 263
Vector 57

BxDF subclass that overrides the BxDF::Sample_f() method also override the BxDF::Pdf() method so that the two return consistent results.

To actually evaluate the PDF for the cosine-weighted sampling method (which we showed earlier was $p(\omega) = \cos\theta/\pi$), it is first necessary to check that ω_o and ω_i lie on the same side of the surface; if not, the sampling probability is zero. Otherwise, the method computes $|\mathbf{n} \cdot \omega_i|$. One potential pitfall with this method is that the order of the ω_o and ω_i arguments is significant. For the cosine-weighted distribution here, for example, $p(\omega_o) \neq p(\omega_i)$ in general. Code that calls this method must be careful to use the correct argument ordering.

⟨*BSDF Inline Functions*⟩ +≡
```
inline bool SameHemisphere(const Vector &w, const Vector &wp) {
    return w.z * wp.z > 0.f;
}
```

⟨*BxDF Method Definitions*⟩ +≡
```
float BxDF::Pdf(const Vector &wo, const Vector &wi) const {
    return SameHemisphere(wo, wi) ? AbsCosTheta(wi) * INV_PI : 0.f;
}
```

This sampling method works well for Lambertian BRDFs, and it works well for the Oren–Nayar model as well, so we will not override it for those classes.

14.5.1 SAMPLING THE BLINN MICROFACET DISTRIBUTION

BRDFs based on microfacet distribution functions are more difficult to sample. For these models, the BRDF is a product of three terms, D, G, and F, which is then divided by two cosine terms; recall Equation (8.8). It is impractical to find the probability distribution that matches the complete model, so instead we will derive a method for drawing samples from the distribution described by the microfacet distribution function D alone. This is an effective strategy for importance sampling the complete model, since the D term accounts for most of its variation.

Therefore, all MicrofacetDistribution implementations must implement methods for sampling from their distribution and computing the value of their PDF, each with the same signature as the corresponding BxDF function, except that their Sample_f() methods return void.

AbsCosTheta() 426
BxDF 428
BxDF::Pdf() 695
BxDF::Sample_f() 694
INV_PI 1002
Microfacet 454
MicrofacetDistribution 454
SameHemisphere() 695
Vector 57

⟨*MicrofacetDistribution Interface*⟩ +≡ 454
```
virtual void Sample_f(const Vector &wo, Vector *wi,
                      float u1, float u2, float *pdf) const = 0;
virtual float Pdf(const Vector &wo, const Vector &wi) const = 0;
```

The Microfacet BxDF, then, forwards the sampling and PDF requests to the distribution function that it holds a pointer to.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum Microfacet::Sample_f(const Vector &wo, Vector *wi,
                              float u1, float u2, float *pdf) const {
    distribution->Sample_f(wo, wi, u1, u2, pdf);
    if (!SameHemisphere(wo, *wi)) return Spectrum(0.f);
    return f(wo, *wi);
}
```

⟨*BxDF Method Definitions*⟩ +≡
```
float Microfacet::Pdf(const Vector &wo, const Vector &wi) const {
    if (!SameHemisphere(wo, wi)) return 0.f;
    return distribution->Pdf(wo, wi);
}
```

Recall that Blinn's microfacet distribution function is $D(\cos\theta_h) = (n+2)(\cos\theta_h)^n$, where $\cos\theta_\mathrm{h} = |\mathbf{n}\cdot\omega_\mathrm{h}|$ and the distribution in terms of directions on the unit sphere is $D(\omega_\mathrm{h}) = ((n+2)/2\pi)(\cos\theta_\mathrm{h})^n$ (Equation (8.9)). Thus, in order to sample incident directions, we will first sample a half-angle direction ω_h according to this distribution, which we will then use to find ω_i by reflecting ω_o about the sampled direction ω_h.

⟨*BxDF Method Definitions*⟩ +≡
```
void Blinn::Sample_f(const Vector &wo, Vector *wi, float u1, float u2,
                     float *pdf) const {
    ⟨Compute sampled half-angle vector ωh for Blinn distribution 697⟩
    ⟨Compute incident direction by reflecting about ωh 697⟩
    ⟨Compute PDF for ωi from Blinn distribution 699⟩
    *pdf = blinn_pdf;
}
```

Because the value of ϕ doesn't affect D, the PDF for this distribution, $p_\mathrm{h}(\theta,\phi)$, is separable into $p_\mathrm{h}(\theta)$ and $p_\mathrm{h}(\phi)$. $p_\mathrm{h}(\phi)$ is constant with a value of $1/(2\pi)$, and thus ϕ values can be sampled by

$$\phi = 2\pi\xi_2.$$

By construction, microfacet distribution functions are already normalized. However, recall that the normalization in Section 8.4.3 was with respect to a projected solid angle, in order to ensure that the area of the microfacets was physically-plausible. Here, we want a normalized distribution of directions on the hemisphere. Following the usual approach, this can be shown to be

$$p_\mathrm{h}(\cos\theta_\mathrm{h}) = (n+1)\cos^n\theta_\mathrm{h}. \quad [14.1]$$

This is a power distribution in $\cos\theta_\mathrm{h}$, and thus a uniform random variable can be used to generate a sample by

$$\cos\theta_\mathrm{h} = \sqrt[n+1]{\xi_1}.$$

(Recall Equation (13.4).)

Blinn 456
Microfacet 454
Microfacet::f() 455
MicrofacetDistribution::Pdf() 695
MicrofacetDistribution::Sample_f() 695
SameHemisphere() 695
Spectrum 263
Vector 57

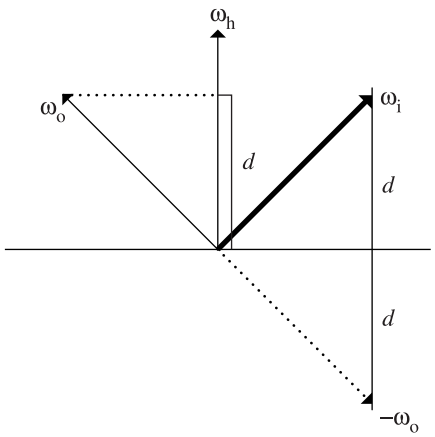


Figure 14.6: The reflection of a direction ω_o about the direction ω_h can be computed by first taking the offset $-\omega_o$ from the origin, giving the vector beneath the surface. We then add two times the distance d, which is given by the projection of ω_o onto ω_h (which is given by their dot product) to give the direction ω_i above the surface.

Because `pbrt` transforms the normal to (0, 0, 1) in the reflection coordinate system, we can almost use the computed direction from spherical coordinates directly. The only additional detail to handle is that if ω_o is in the opposite hemisphere than the normal, then the half-angle vector needs to be flipped to be in that hemisphere as well.

⟨*Compute sampled half-angle vector ω_h for Blinn distribution*⟩ ≡ 696
```
float costheta = powf(u1, 1.f / (exponent+1));
float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
float phi = u2 * 2.f * M_PI;
Vector wh = SphericalDirection(sintheta, costheta, phi);
if (!SameHemisphere(wo, wh)) wh = -wh;
```

All that's left to do is to apply the formula for reflection of a vector about another vector (Figure 14.6):

⟨*Compute incident direction by reflecting about ω_h*⟩ ≡ 696, 699
```
*wi = -wo + 2.f * Dot(wo, wh) * wh;
```

There's an important detail to take care of to compute the value of the PDF for the sampled direction, however. The microfacet distribution gives the distribution of normals around the *half-angle vector*, but the reflection integral is with respect to the *incoming vector*. These distributions are not the same, and we must convert the half-angle PDF to the incoming angle PDF. In other words, we must change from a density in terms of ω_h to one in terms of ω_i using the techniques introduced in Section 13.5. Doing so requires applying the adjustment for a change of variables $d\omega_h/d\omega_i$.

`M_PI` 1002
`SameHemisphere()` 695
`SphericalDirection()` 292
`Vector` 57

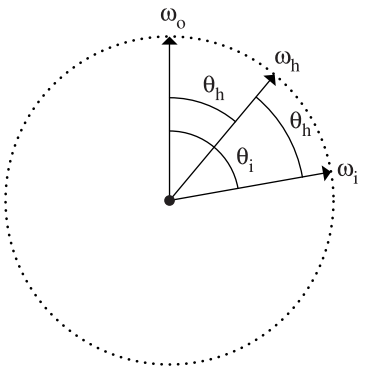


Figure 14.7: The adjustment for change of variable from sampling from the half-angle distribution to sampling from the incident direction distribution can be derived with an observation about the relative angles involved: $\theta_i = 2\theta_h$.

A simple geometric construction gives the relationship between the two distributions. Consider the spherical coordinate system oriented about ω_o (Figure 14.7). The differential solid angles $d\omega_i$ and $d\omega_h$ are $\sin\theta_i\, d\theta_i d\phi_i$ and $\sin\theta_h\, d\theta_h d\phi_h$, respectively; thus,

$$\frac{d\omega_h}{d\omega_i} = \frac{\sin\theta_h\, d\theta_h\, d\phi_h}{\sin\theta_i\, d\theta_i\, d\phi_i}.$$

Because ω_i is computed by reflecting ω_o about ω_h, $\theta_i = 2\theta_h$. Furthermore, because $\phi_i = \phi_h$, we can find the desired conversion factor:

$$\begin{aligned}\frac{d\omega_h}{d\omega_i} &= \frac{\sin\theta_h\, d\theta_h\, d\phi_h}{\sin 2\theta_h\, 2\, d\theta_h\, d\phi_h} \\ &= \frac{\sin\theta_h}{4\cos\theta_h \sin\theta_h} \\ &= \frac{1}{4\cos\theta_h} \\ &= \frac{1}{4(\omega_i \cdot \omega_h)} = \frac{1}{4(\omega_o \cdot \omega_h)}.\end{aligned}$$

Therefore, the PDF after transformation is

$$p(\theta) = \frac{p_h(\theta)}{4(\omega_o \cdot \omega_h)}.$$

Using the definition of the Blinn distribution over the hemisphere from Equation (14.1), we have:

⟨*Compute PDF for ω_i from Blinn distribution*⟩ ≡ 696, 699
```
float blinn_pdf = ((exponent + 1.f) * powf(costheta, exponent)) /
                  (2.f * M_PI * 4.f * Dot(wo, wh));
if (Dot(wo, wh) <= 0.f) blinn_pdf = 0.f;
```

We can reuse this fragment to implement the `Blinn::Pdf()` method.

⟨*BxDF Method Definitions*⟩ +≡
```
float Blinn::Pdf(const Vector &wo, const Vector &wi) const {
    Vector wh = Normalize(wo + wi);
    float costheta = AbsCosTheta(wh);
    ⟨Compute PDF for ωi from Blinn distribution 699⟩
    return blinn_pdf;
}
```

14.5.2 SAMPLING THE ANISOTROPIC MICROFACET MODEL

Ashikhmin and Shirley (2000, 2002) gave the equations for sampling from the distribution defined by their anisotropic BRDF model. We will restate their results here without deriving them; interested readers should consult the original publications for details and derivations.

⟨*BxDF Method Definitions*⟩ +≡
```
void Anisotropic::Sample_f(const Vector &wo, Vector *wi,
                           float u1, float u2, float *pdf) const {
    ⟨Sample from first quadrant and remap to hemisphere to sample ωh 699⟩
    ⟨Compute incident direction by reflecting about ωh 697⟩
    ⟨Compute PDF for ωi from anisotropic distribution 701⟩
    *pdf = anisotropic_pdf;
}
```

The sampling method in the `Anisotropic::sampleFirstQuadrant()` function samples a direction in the first quadrant of the unit hemisphere—that is, with spherical angles in the range $(\theta, \phi) \in [0, \pi/2] \times [0, \pi/2]$. It's easier to derive a sampling method for this distribution by considering only that part of the problem and using it as a building block for sampling all directions. Here, we use ξ_1 to select a quadrant, remap it to $[0, 1]$ to generate a sample from the first quadrant, and then map the sample to the desired quadrant. The remapping of ξ_1 and the sampled ϕ value are all done carefully to preserve stratification: given a small difference in ξ_1 that moves from one quadrant to another, the generated half-angle direction should change only slightly.

⟨*Sample from first quadrant and remap to hemisphere to sample ω_h*⟩ ≡ 699
```
float phi, costheta;
if (u1 < .25f) {
    sampleFirstQuadrant(4.f * u1, u2, &phi, &costheta);
} else if (u1 < .5f) {
    u1 = 4.f * (.5f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = M_PI - phi;
```

AbsCosTheta() 426
Anisotropic 458
Anisotropic::sampleFirstQuadrant() 700
Blinn 456
Blinn::exponent 456
Blinn::Pdf() 699
Dot() 60
M_PI 1002
SameHemisphere() 695
SphericalDirection() 292
Vector 57
Vector::Normalize() 63

```
} else if (u1 < .75f) {
    u1 = 4.f * (u1 - .5f);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi += M_PI;
} else {
    u1 = 4.f * (1.f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = 2.f * M_PI - phi;
}
float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
Vector wh = SphericalDirection(sintheta, costheta, phi);
if (!SameHemisphere(wo, wh)) wh = -wh;
```

The half-angle vector in the first quadrant of the hemisphere can be sampled from this distribution function by

$$\phi = \arctan\left(\sqrt{\frac{e_x+1}{e_y+1}}\tan\left(\frac{\pi\xi_1}{2}\right)\right)$$

$$\cos\theta = \xi_2^{(e_x\cos^2\phi + e_y\sin^2\phi+1)^{-1}}.$$

Note that, if $e_x = e_y$ and the microfacet distribution is isotropic, this gives the same sampling technique as was used for the earlier Blinn distribution.

⟨*BxDF Method Definitions*⟩ +≡

```
void Anisotropic::sampleFirstQuadrant(float u1, float u2,
        float *phi, float *costheta) const {
    if (ex == ey)
        *phi = M_PI * u1 * 0.5f;
    else
        *phi = atanf(sqrtf((ex+1.f) / (ey+1.f)) *
                     tanf(M_PI * u1 * 0.5f));
    float cosphi = cosf(*phi), sinphi = sinf(*phi);
    *costheta = powf(u2, 1.f/(ex * cosphi * cosphi +
                              ey * sinphi * sinphi + 1));
}
```

Similar to the Blinn case, the PDF for a direction sampled with this method is the value of the normalized distribution over the hemisphere of directions, although as before we must account for the change of variables required to convert from the half-angle distribution to the incident angle distribution just as was done in `Blinn::Pdf()`.

⟨*BxDF Method Definitions*⟩ +≡

```
float Anisotropic::Pdf(const Vector &wo, const Vector &wi) const {
    Vector wh = Normalize(wo + wi);
    ⟨Compute PDF for ωi from anisotropic distribution 701⟩
    return anisotropic_pdf;
}
```

Anisotropic 458
Anisotropic::ex 458
Anisotropic::ey 458
Blinn::Pdf() 699
M_PI 1002
Vector 57
Vector::Normalize() 63

⟨*Compute PDF for ω_i from anisotropic distribution*⟩ ≡ 699, 700
```
float costhetah = AbsCosTheta(wh);
float ds = 1.f - costhetah * costhetah;
float anisotropic_pdf = 0.f;
if (ds > 0.f && Dot(wo, wh) > 0.f) {
    float e = (ex * wh.x * wh.x + ey * wh.y * wh.y) / ds;
    float d = sqrtf((ex+1.f) * (ey+1.f)) * INV_TWOPI *
              powf(costhetah, e);
    anisotropic_pdf = d / (4.f * Dot(wo, wh));
}
```

14.5.3 SAMPLING FresnelBlend

The FresnelBlend class is a mixture of a diffuse and glossy term. A straightforward approach to sampling this BRDF is to sample from both a cosine-weighted distribution as well as the microfacet distribution. The implementation here chooses between the two with equal probability based on whether ξ_1 is less than or greater than 0.5. In both cases, it remaps ξ_1 to cover the range $[0, 1]$ after using it to make this decision. (Otherwise, values of ξ_1 used for the cosine-weighted sampling would always be less than 0.5, for example.) Using the sample ξ_1 for two purposes in this manner slightly reduces the quality of the stratification of the (ξ_1, ξ_2) values that are actually used for sampling directions, although this is less of a shortcoming than randomly selecting between the two sampling methods with a new unstratified random number.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum FresnelBlend::Sample_f(const Vector &wo, Vector *wi,
                                float u1, float u2, float *pdf) const {
    if (u1 < .5) {
        u1 = 2.f * u1;
        ⟨Cosine-sample the hemisphere, flipping the direction if necessary 694⟩
    }
    else {
        u1 = 2.f * (u1 - .5f);
        distribution->Sample_f(wo, wi, u1, u2, pdf);
        if (!SameHemisphere(wo, *wi)) return Spectrum(0.f);
    }
    *pdf = Pdf(wo, *wi);
    return f(wo, *wi);
}
```

The PDF for this sampling strategy is simple; it is just an average of the two PDFs used.

⟨*BxDF Method Definitions*⟩ +≡
```
float FresnelBlend::Pdf(const Vector &wo, const Vector &wi) const {
    if (!SameHemisphere(wo, wi)) return 0.f;
    return .5f * (AbsCosTheta(wi) * INV_PI + distribution->Pdf(wo, wi));
}
```

AbsCosTheta() 426
FresnelBlend 460
INV_PI 1002
INV_TWOPI 1002
MicrofacetDistribution::Pdf() 695
MicrofacetDistribution::Sample_f() 695
Spectrum 263
Vector 57

14.5.4 SPECULAR REFLECTION AND TRANSMISSION

The Dirac delta distributions that were previously used to define the BRDF for specular reflection and the BTDF for specular transmission fit into this sampling framework well, as long as a few conventions are kept in mind when using their sampling and PDF functions.

Recall that the Dirac delta distribution is defined such that

$$\delta(x) = 0 \text{ for all } x \neq 0$$

and

$$\int_{-\infty}^{\infty} \delta(x)\, \mathrm{d}x = 1.$$

Thus, it is a probability density function, where the PDF has a value of zero for all $x \neq 0$. Generating a sample from such a distribution is trivial; there is only one possible value for it to take on. When thought of in this way, the implementations of `Sample_f()` for the `SpecularReflection` and `SpecularTransmission` `BxDF`s can be seen to fit naturally into the Monte Carlo sampling framework.

It is not as simple to determine which value should be returned for the value of the PDF. Strictly speaking, the delta distribution is not a true function, but must be defined as the limit of another function—for example, one describing a box of unit area whose width approaches zero; see Chapter 5 of Bracewell (2000) for further discussion and references. Thought of in this way, the value of $\delta(0)$ tends toward infinity. Certainly, returning an infinite or very large value for the PDF is not going to lead to correct results from the renderer.

Recall that BSDFs defined with delta components also have these delta components in their f_r functions, a detail that was glossed over when we returned values from their `Sample_f()` methods in Chapter 8. Thus, the Monte Carlo estimator for the scattering equation with such a BSDF is written

$$\frac{1}{N}\sum_{i}^{N} \frac{f_\mathrm{r}(\mathrm{p}, \omega_\mathrm{o}, \omega_i) L_\mathrm{i}(\mathrm{p}, \omega_i) |\cos\theta_i|}{p(\omega_i)} = \frac{1}{N}\sum_{i}^{N} \frac{\rho_\mathrm{hd}(\omega_\mathrm{o}) \frac{\delta(\omega - \omega_i)}{|\cos\theta_i|} L_\mathrm{i}(\mathrm{p}, \omega_i) |\cos\theta_i|}{p(\omega_i)},$$

where $\rho_\mathrm{hd}(\omega_\mathrm{o})$ is the hemispherical–directional reflectance and ω is the direction for perfect specular reflection or transmission.

Because the PDF $p(\omega_i)$ has a delta term as well, $p(\omega_i) = \delta(\omega - \omega_i)$, the two delta distributions cancel out, and the estimator is

$$\rho_\mathrm{hd}(\omega_\mathrm{o}) L_\mathrm{i}(\mathrm{p}, \omega),$$

exactly the quantity computed by the Whitted integrator, for example.

Therefore, the implementations here return a constant value of one for the PDF for specular reflection and transmission when sampled using `Sample_f()`, with the convention that for specular `BxDF`s there is an implied delta distribution in the PDF value that is expected to cancel out with the implied delta distribution in the value of the BSDF when the estimator is evaluated. The respective `Pdf()` methods therefore return zero for all di-

BxDF 428
SpecularReflection 440
SpecularTransmission 444

rections, since there is zero probability that another sampling method will randomly find the direction from a delta distribution.

⟨*SpecularReflection Public Methods*⟩ +≡ 440
```
float Pdf(const Vector &wo, const Vector &wi) const {
    return 0.;
}
```

⟨*SpecularTransmission Public Methods*⟩ +≡ 444
```
float Pdf(const Vector &wo, const Vector &wi) const {
    return 0.;
}
```

There is a potential pitfall with this convention: when multiple importance sampling is used to compute weights, PDF values that include these implicit delta distributions can't be freely mixed with regular PDF values. This isn't a problem in practice, since there's no reason to apply MIS when there's a delta distribution in the integrand. The light transport routines in the next chapter have appropriate logic to be sure to avoid this error.

14.5.5 APPLICATION: ESTIMATING REFLECTANCE

As an example of their application, we will show how these BxDF sampling routines can be used in computing estimates of the reflectance integrals defined in Section 8.1.1 for arbitrary BRDFs. For example, recall that the hemispherical–directional reflectance is

$$\rho_{\mathrm{hd}}(\omega_{\mathrm{o}}) = \int_{\mathcal{H}^2(\mathbf{n})} f_{\mathrm{r}}(\omega_{\mathrm{o}}, \omega_{\mathrm{i}}) \, |\cos \theta_{\mathrm{i}}| \, \mathrm{d}\omega_{\mathrm{i}}.$$

Recall from Section 8.1.1 that BxDF::rho() method implementations take two additional parameters, nSamples and samples; here, we can see how they are used for Monte Carlo sampling. For BxDF implementations that can't compute the reflectance in closed form, the nSamples parameter specifies the number of Monte Carlo samples to take, and sample values themselves are provided in the samples array. The samples array should have 2*nSamples entries, where the first two values provide the first 2D sample value and so forth.

The generic BxDF::rho() method computes a Monte Carlo estimate of this value for any BxDF, using the provided samples and taking advantage of the BxDF's sampling method to compute the estimate with importance sampling.

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum BxDF::rho(const Vector &w, int nSamples,
                   const float *samples) const {
    Spectrum r = 0.;
    for (int i = 0; i < nSamples; ++i) {
        ⟨Estimate one term of ρhd 704⟩
    }
    return r / float(nSamples);
}
```

BxDF 428
BxDF::rho() 430
Spectrum 263
Vector 57

Actually evaluating the estimator is a matter of sampling the reflection function's distribution, finding its value, and dividing it by the value of the PDF. Each term of the estimator

$$\frac{1}{N}\sum_{j}^{N}\frac{f_{\mathrm{r}}(\omega,\omega_j)\,|\cos\theta_j|}{p(\omega_j)}$$

is easily evaluated. The BxDF's Sample_f() method returns all of ω_j, $p(\omega_j)$ and the value of $f_{\mathrm{r}}(\omega_{\mathrm{o}},\omega_j)$. The only tricky part is when $p(\omega_j)=0$, which must be detected here, since otherwise a division by zero would place an infinite value in r.

⟨*Estimate one term of* ρ_{hd}⟩ ≡ 703
```
Vector wi;
float pdf = 0.f;
Spectrum f = Sample_f(w, &wi, samples[2*i], samples[2*i+1], &pdf);
if (pdf > 0.) r += f * AbsCosTheta(wi) / pdf;
```

The hemispherical–hemispherical reflectance can be estimated similarly. Given

$$\rho_{\mathrm{hh}}=\frac{1}{\pi}\int_{\mathcal{H}^2(\mathbf{n})}\int_{\mathcal{H}^2(\mathbf{n})}f_{\mathrm{r}}(\omega',\omega'')\,|\cos\theta'\cos\theta''|\,\mathrm{d}\omega'\,\mathrm{d}\omega'',$$

two vectors, ω' and ω'', must be sampled for each term of the estimate

$$\frac{1}{N}\sum_{j}^{N}\frac{f_{\mathrm{r}}(\omega'_j,\omega''_j)\,|\cos\theta'_j\cos\theta''_j|}{p(\omega'_j)\,p(\omega''_j)}.$$

⟨*BxDF Method Definitions*⟩ +≡
```
Spectrum BxDF::rho(int nSamples, const float *samples1,
                   const float *samples2) const {
    Spectrum r = 0.;
    for (int i = 0; i < nSamples; ++i) {
        ⟨Estimate one term of ρhh 704⟩
    }
    return r / (M_PI*nSamples);
}
```

The implementation here samples the first direction ω' uniformly over the hemisphere. Given this, the second direction can be sampled with the BxDF::Sample_f() method.[1]

⟨*Estimate one term of* ρ_{hh}⟩ ≡ 704
```
Vector wo, wi;
wo = UniformSampleHemisphere(samples1[2*i], samples1[2*i+1]);
float pdf_o = INV_TWOPI, pdf_i = 0.f;
Spectrum f = Sample_f(wo, &wi, samples2[2*i], samples2[2*i+1], &pdf_i);
if (pdf_i > 0.)
    r += f * AbsCosTheta(wi) * AbsCosTheta(wo) / (pdf_o * pdf_i);
```

AbsCosTheta() 426
BxDF 428
BxDF::Sample_f() 694
INV_TWOPI 1002
M_PI 1002
Spectrum 263
UniformSampleHemisphere() 664
Vector 57

1 It could be argued that a shortcoming of the BxDF sampling interface is that there aren't entry points to sample from the four-dimensional distribution of $f_{\mathrm{r}}(\mathrm{p},\omega,\omega')$. This is a reasonably esoteric case for the applications envisioned for pbrt, however.

14.5.6 SAMPLING BSDFs

Given these methods to sample individual BxDFs, we can now define a sampling method for the BSDF class, BSDF::Sample_f(). This method is called by SurfaceIntegrators when they want to sample according to the BSDF's distribution; it calls the individual BxDF::Sample_f() methods to generate samples. The BSDF stores pointers to one or more individual BxDFs that can be sampled individually, but here we will sample from the density that is the sum of their individual densities,

$$p(\omega) = \frac{1}{N} \sum_{i}^{N} p_i(\omega).$$

(Exercise 14.1 at the end of the chapter discusses the alternative of sampling from the BxDFs according to probabilities based on their respective reflectances; this approach can be more efficient if they vary significantly.)

The BSDF::Sample_f() method takes three random variables to drive this process; one is used to select among the BxDFs, and the other two to pass along to the particular sampling method chosen. These sample values are encapsulated by the BSDFSample structure.

⟨*BSDF Declarations*⟩ +≡
```
struct BSDFSample {
    ⟨BSDFSample Public Methods 705⟩
    float uDir[2], uComponent;
};
```

The BSDFSample can be initialized by directly passing in values for the three samples.

⟨*BSDFSample Public Methods*⟩ ≡ 705
```
BSDFSample(float up0, float up1, float ucomp) {
    uDir[0] = up0;
    uDir[1] = up1;
    uComponent = ucomp;
}
```

Alternatively, a convenience method takes a pseudo-random number generator and initializes the samples using it.

⟨*BSDFSample Public Methods*⟩ +≡ 705
```
BSDFSample(RNG &rng) {
    uDir[0] = rng.RandomFloat();
    uDir[1] = rng.RandomFloat();
    uComponent = rng.RandomFloat();
}
```

Most commonly, we'll want to initialize the BSDFSample with values from a Sample object filled in by the Sampler. The BSDFSampleOffsets structure helps with the bookkeeping for this case.

BSDF 478
BSDF::Sample_f() 706
BSDFSample 705
BSDFSample::uComponent 705
BSDFSample::uDir 705
BSDFSampleOffsets 706
BxDF 428
BxDF::Sample_f() 694
RNG 1003
RNG::RandomFloat() 1003
Sample 343
Sampler 340
SurfaceIntegrator 740

⟨*BSDF Declarations*⟩ +≡
```
struct BSDFSampleOffsets {
    int nSamples, componentOffset, dirOffset;
};
```

Its constructor takes the total number of samples to be taken and the `Sample` object from the `Sampler`; it calls the `Add1D()` method to request 1D samples for choosing a `BxDF` component, and the `Add2D()` method to request 2D samples for the sampled direction.

⟨*BSDF Method Definitions*⟩ +≡
```
BSDFSampleOffsets::BSDFSampleOffsets(int count, Sample *sample) {
    nSamples = count;
    componentOffset = sample->Add1D(nSamples);
    dirOffset = sample->Add2D(nSamples);
}
```

With the `BSDFSampleOffsets` structure in hand, we can implement a third constructor for the `BSDFSample` class. It takes a filled-in `Sample`, a previously initialized `BSDFSampleOffsets`, and the sample number to be taken. (`num` should be less than the `count` value passed to the `BSDFSampleOffsets` constructor). It then takes care of extracting the appropriate values from the `Sample`'s sample values and initializing the member variables.

⟨*BSDF Method Definitions*⟩ +≡
```
BSDFSample::BSDFSample(const Sample *sample,
                       const BSDFSampleOffsets &offsets, uint32_t n) {
    uDir[0] = sample->twoD[offsets.dirOffset][2*n];
    uDir[1] = sample->twoD[offsets.dirOffset][2*n+1];
    uComponent = sample->oneD[offsets.componentOffset][n];
}
```

With the definition of `BSDFSample` in hand, the `BSDF::Sample_f()` method can now be implemented. The outgoing direction passed to it and the incident direction it returns are in world space, which is indicated by the `W` at the end of the respective variable names.

⟨*BSDF Method Definitions*⟩ +≡
```
Spectrum BSDF::Sample_f(const Vector &woW, Vector *wiW,
                        const BSDFSample &bsdfSample, float *pdf,
                        BxDFType flags, BxDFType *sampledType) const {
    ⟨Choose which BxDF to sample 707⟩
    ⟨Sample chosen BxDF 707⟩
    ⟨Compute overall PDF with all matching BxDFs 707⟩
    ⟨Compute value of BSDF for sampled direction 708⟩
}
```

This method first determines which `BxDF`'s sampling method to use for this particular sample. This is complicated by the fact that the caller may pass in flags that the chosen `BxDF` must match (e.g., specifying that only diffuse components should be considered). Thus, only a subset of the sampling densities may actually be used here.

BSDF 478
BSDFSample 705
BSDFSample::uComponent 705
BSDFSample::uDir 705
BSDFSampleOffsets 706
BSDFSampleOffsets::componentOffset 706
BSDFSampleOffsets::dirOffset 706
BxDF 428
BxDFType 428
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sample::oneD 344
Sample::twoD 344
Sampler 340
Spectrum 263
Vector 57

⟨*Choose which* BxDF *to sample*⟩ ≡ 706
```
int matchingComps = NumComponents(flags);
if (matchingComps == 0) {
    *pdf = 0.f;
    return Spectrum(0.f);
}
int which = min(Floor2Int(bsdfSample.uComponent * matchingComps),
                matchingComps-1);
BxDF *bxdf = NULL;
int count = which;
for (int i = 0; i < nBxDFs; ++i)
    if (bxdfs[i]->MatchesFlags(flags) && count-- == 0) {
        bxdf = bxdfs[i];
        break;
    }
```

Once the appropriate BxDF is chosen, its BxDF::Sample_f() method is called. Recall that these methods expect and return vectors in the BxDF's local coordinate system, so the supplied vector must be transformed to the BxDF's coordinate system and the returned vector must be transformed back into world coordinates.

⟨*Sample chosen* BxDF⟩ ≡ 706
```
Vector wo = WorldToLocal(woW);
Vector wi;
*pdf = 0.f;
Spectrum f = bxdf->Sample_f(wo, &wi, bsdfSample.uDir[0],
                            bsdfSample.uDir[1], pdf);
if (*pdf == 0.f)
    return 0.f;
if (sampledType) *sampledType = bxdf->type;
*wiW = LocalToWorld(wi);
```

To compute the actual PDF for sampling the direction ω_i, we need the average of all of the PDFs of the BxDFs that could have been used, given the BxDFType flags passed in. Because *pdf already holds the PDF value for the distribution the sample was taken from, we only need to add in the contributions of the others. It's important that this step be skipped if the chosen BxDF is perfectly specular, since the PDF has an implicit delta distribution in it. It would be incorrect to add the other PDF values to this one, since it is a delta term represented with value one, rather than as an actual delta distribution.

⟨*Compute overall PDF with all matching* BxDFs⟩ ≡ 706
```
if (!(bxdf->type & BSDF_SPECULAR) && matchingComps > 1)
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i] != bxdf && bxdfs[i]->MatchesFlags(flags))
            *pdf += bxdfs[i]->Pdf(wo, wi);
if (matchingComps > 1) *pdf /= matchingComps;
```

Given the sampled direction, this method needs to compute the value of the BSDF for the pair of directions (ω_o, ω_i) accounting for all of the relevant components in the

BSDF::bxdfs 479
BSDF::LocalToWorld() 480
BSDF::NumComponents() 479
BSDF::WorldToLocal() 480
BSDF_SPECULAR 428
BxDF 428
BxDF::MatchesFlags() 429
BxDF::Pdf() 695
BxDF::Sample_f() 694
BxDF::type 429
BxDFType 428
Floor2Int() 1002
Spectrum 263
Vector 57

BSDF, unless the sampled direction was from a specular component, in which case the value returned from `Sample_f()` earlier is used. (If a specular component generated this direction, its `BxDF::f()` method will return black, even if we pass back the direction its sampling routine returned.)

While this method could just call the `BSDF::f()` method to compute the BSDF's value, the value can be more efficiently computed by calling the `BxDF::f()` methods directly, taking advantage of the fact that here we already have the directions in both world space and the reflection coordinate system available.

⟨*Compute value of BSDF for sampled direction*⟩ ≡ 706

```
if (!(bxdf->type & BSDF_SPECULAR)) {
    f = 0.;
    if (Dot(*wiW, ng) * Dot(woW, ng) > 0) // ignore BTDFs
        flags = BxDFType(flags & ~BSDF_TRANSMISSION);
    else // ignore BRDFs
        flags = BxDFType(flags & ~BSDF_REFLECTION);
    for (int i = 0; i < nBxDFs; ++i)
        if (bxdfs[i]->MatchesFlags(flags))
            f += bxdfs[i]->f(wo, wi);
}
return f;
```

The `BSDF::Pdf()` method does a similar computation, looping over the `BxDF`s and calling their `Pdf()` methods to find the PDF for an arbitrary sampled direction:

⟨*BSDF Public Methods*⟩ +≡ 478

```
float Pdf(const Vector &wo, const Vector &wi,
          BxDFType flags = BSDF_ALL) const;
```

14.6 SAMPLING LIGHT SOURCES

Because direct illumination from light sources makes a key contribution to reflected light at a point, it is important to be able to take a point and sample those directions around it where the direct illumination may take on a nonzero value. Consider a diffuse surface illuminated by a small spherical area light source (Figure 14.8): sampling directions using the BSDF's sampling distribution is likely to be very inefficient because the light is only visible along a small cone of directions from the point.

A much better approach is to instead use a sampling distribution that is based on the light source. For example, the sampling routine could choose from among only those directions where the sphere is potentially visible. The sphere may actually be occluded for some or all of these directions, though it is difficult to account for this issue in defining the sampling densities.

This section will apply the now familiar Monte Carlo machinery to the light sampling problem. Furthermore, it will show the derivations and implementations of techniques for sampling rays leaving light sources. Being able to do this is crucial for bidirectional light transport algorithms like photon mapping and bidirectional path tracing.

BSDF::f() 481
BSDF::Pdf() 708
BSDF_ALL 428
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDF 428
BxDF::f() 429
BxDF::MatchesFlags() 429
BxDFType 428
Vector 57

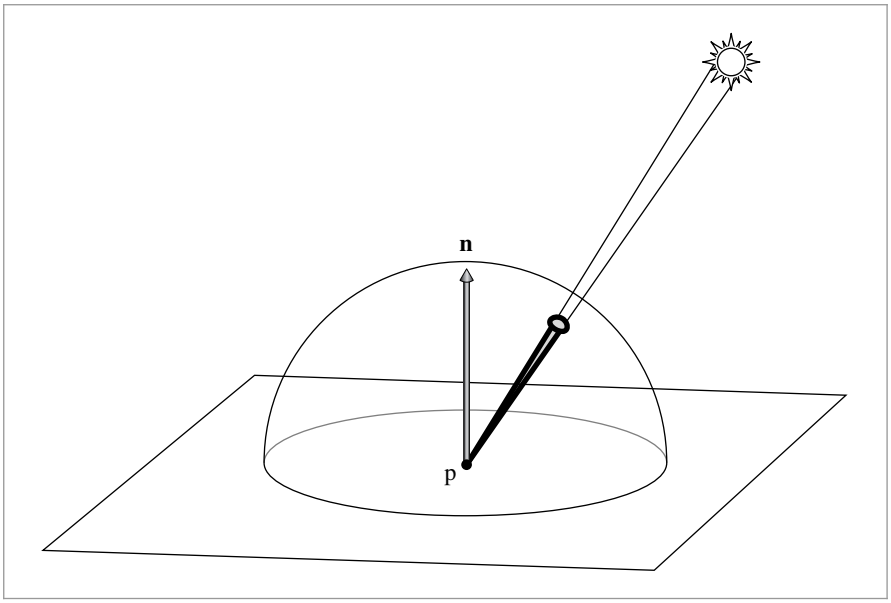


Figure 14.8: Another effective sampling strategy for choosing an incident direction from a point p for direct lighting computations is to use the light source to define a distribution of directions with respect to solid angle at the point. Here, a small spherical light source is illuminating the point. The cone of directions that the sphere subtends is a much better sampling distribution to use than a uniform distribution, for example.

A number of the light transport algorithms in the following chapters need to randomly sample a light source based on its power; for efficiency, it's generally worthwhile to take more samples from light sources that have a greater effect on the illumination in the scene. Sampling lights by their emitted power is generally a good proxy for this. The `ComputeLightSamplingCDF()` utility function creates a `Distribution1D` based on the power of all of the lights in the scene to compute a cumulative distribution function for sampling from the lights.

⟨*Integrator Utility Functions*⟩ +≡

```
Distribution1D *ComputeLightSamplingCDF(const Scene *scene) {
    uint32_t nLights = int(scene->lights.size());
    vector<float>lightPower(nLights, 0.f);
    for (uint32_t i = 0; i < nLights; ++i)
        lightPower[i] = scene->lights[i]->Power(scene).y();
    return new Distribution1D(&lightPower[0], nLights);
}
```

BxDF::Pdf() 695
BxDF::Sample_f() 694
Distribution1D 648
Light 606
Light::Pdf() 711
Light::Power() 608
Light::Sample_L() 608
Scene 22
Scene::lights 23
Spectrum::y() 273

14.6.1 BASIC INTERFACE

There are three main methods related to sampling that all `Lights` must implement. Two of them, `Light::Sample_L()` and `Light::Pdf()`, are comparable to the `BxDF::Sample_f()` and `BxDF::Pdf()` methods; they also generate samples from a distribution of directions

over the sphere from a point in the scene. Here, the distribution doesn't try to match the BSDF at the point, but instead tries to match the incident radiance distribution from direct illumination from the light source.

The `Light::Sample_L()` method was first introduced in Section 12.1. For reference, here is its declaration:

```
virtual Spectrum Sample_L(const Point &p, float pEpsilon,
    const LightSample &ls, float time, Vector *wi, float *pdf,
    VisibilityTester *vis) const = 0;
```

We can now understand the meaning of its `LightSample` and `pdf` parameters: `LightSample` stores a three-dimensional random sample for sampling the light source (similar to the `BSDFSample`), and the PDF for sampling the chosen direction is returned in `*pdf`. Not all lights need all three of these samples, but for lights that are comprised of many `Shapes`, then we need one sample to select which shape to sample from and two samples to choose a point on its surface. The definition of `LightSample` is almost the same as `BSDFSample`, so we'll elide much of its definition here.

⟨*Light Declarations*⟩ +≡
```
struct LightSample {
    ⟨LightSample Public Methods⟩
    float uPos[2], uComponent;
};
```

Also similar to `BSDFSampleOffsets`, `LightSampleOffsets` handles requesting appropriate sample values from the `Sampler` and then extracting them from a specific `Sample`.

⟨*Light Declarations*⟩ +≡
```
struct LightSampleOffsets {
    LightSampleOffsets(int count, Sample *sample);
    int nSamples, componentOffset, posOffset;
};
```

⟨*Light Method Definitions*⟩ +≡
```
LightSampleOffsets::LightSampleOffsets(int count, Sample *sample) {
    nSamples = count;
    componentOffset = sample->Add1D(nSamples);
    posOffset = sample->Add2D(nSamples);
}
```

⟨*Light Method Definitions*⟩ +≡
```
LightSample::LightSample(const Sample *sample,
        const LightSampleOffsets &offsets, uint32_t n) {
    uPos[0] = sample->twoD[offsets.posOffset][2*n];
    uPos[1] = sample->twoD[offsets.posOffset][2*n+1];
    uComponent = sample->oneD[offsets.componentOffset][n];
}
```

Code that calls `Light::Sample_L()` should also check the value returned by the light's `IsDeltaLight()` method to see if the light source is described by a delta distribution (e.g.,

BSDFSample 705
BSDFSampleOffsets 706
Light::Sample_L() 608
LightSample 710
LightSample::uComponent 710
LightSample::uPos 710
LightSampleOffsets 710
LightSampleOffsets::
componentOffset 710
LightSampleOffsets::
posOffset 710
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sample::oneD 344
Sample::twoD 344
Sampler 340

a point light). There are similar issues related to how to handle PDF values returned from lights described by delta distributions as there are for perfectly specular BSDFs (recall the discussion in Section 14.5.4).

The `Light`'s `Pdf()` method returns the probability density with respect to solid angle for the light's `Sample_L()` method sampling the direction `wi` from the point `p`.

⟨*Light Interface*⟩ +≡ 606
```
virtual float Pdf(const Point &p, const Vector &wi) const = 0;
```

The third light sampling method is quite different. It samples a ray from a distribution of rays *leaving* the light, returning the ray in `*ray` and the surface normal at the point on the light source in `*Ns`. Being able to sample rays in this manner is necessary to support algorithms that construct paths of light leaving the light source.

The returned PDF value should be expressed in terms of the product of the density with respect to surface area on the light and the density with respect to solid angle. The implementations of this method in this section will sample the two distributions separately and compute their product, rather than trying to construct and sample a joint four-dimensional distribution. Implementations of this method may need up to five sample values, with the two additional ones used to determine the direction of the ray (after the first three are used to determine its origin on the light source).

⟨*Light Interface*⟩ +≡ 606
```
virtual Spectrum Sample_L(const Scene *scene, const LightSample &ls,
                          float u1, float u2, float time, Ray *ray,
                          Normal *Ns, float *pdf) const = 0;
```

14.6.2 LIGHTS WITH SINGULARITIES

Just as with perfect specular reflection and transmission, light sources that are defined in terms of delta functions fit naturally into this sampling framework, although they require care on the part of the routines that call their sampling methods, since there are implicit delta distributions in the radiance and PDF values that they return. For the most part, these delta distributions naturally cancel out when estimators are evaluated, although multiple importance sampling code must be aware of this case, just as was the case with BSDFs.

`Light` 606
`LightSample` 710
`Normal` 65
`Point` 63
`PointLight::Sample_L()` 611
`Ray` 66
`Scene` 22
`Spectrum` 263
`Vector` 57

Point Lights

Point lights are described by a delta distribution such that they only illuminate a receiving point from a single direction. Thus, the sampling problem is trivial. The `PointLight::Sample_L()` method was already implemented in Section 12.2, where we glossed over the nuance that the method performs Monte Carlo sampling of a delta distribution and thus always returns a single direction and doesn't need the random sample values.

Due to the delta distribution, the `PointLight::Pdf()` method returns zero. This reflects the fact that there is no chance for some other sampling process to randomly generate a point on an infinitesimal light source.

⟨*PointLight Method Definitions*⟩ +≡
```
float PointLight::Pdf(const Point &, const Vector &) const {
    return 0.;
}
```

The sampling method for generating rays leaving point lights is also straightforward. The origin of the ray must be the light's position; this part of the density is described with a delta distribution. Directions are uniformly sampled over the sphere, and the overall sampling density is the product of these two densities. As usual, we'll ignore the delta distribution that is in the actual PDF because it is canceled out by a (missing) corresponding delta term in the radiance value in the `Spectrum` returned by the sampling routine.

⟨*PointLight Method Definitions*⟩ +≡
```
Spectrum PointLight::Sample_L(const Scene *scene, const LightSample &ls,
        float u1, float u2, float time, Ray *ray, Normal *Ns,
        float *pdf) const {
    *ray = Ray(lightPos, UniformSampleSphere(ls.uPos[0], ls.uPos[1]),
               0.f, INFINITY, time);
    *Ns = (Normal)ray->d;
    *pdf = UniformSpherePdf();
    return Intensity;
}
```

Spotlights

The `SpotLight::Sample_L()` method was also previously implemented in Section 12.2.1; it also returns zero from its `Pdf()` method.

The method for sampling an outgoing ray with a reasonable distribution for the spotlight is more interesting. While it could just sample directions uniformly on the sphere as was done for the point light, this distribution is likely to be a bad match for the spotlight's actual distribution. For example, if the light has a very narrow beam angle, many samples will be taken in directions where the light doesn't cast any illumination. Instead, we will sample from a uniform distribution over the cone of directions in which the light casts illumination. Although the sampling distribution does not try to account for the falloff toward the edges of the beam, this is only a minor shortcoming in practice.

The PDF $p(\theta, \phi)$ for the spotlight's illumination distribution is separable with $p(\phi) = 1/(2\pi)$. We thus just need to find the distribution for θ and to sample from this distribution. We would like to sample a direction θ uniformly over the cone of directions around a central direction up to the maximum angle of the beam, $\theta_{\max}$. Incorporating the $\sin\theta$ term from the measure on the unit sphere from Equation (5.4),

$$\begin{aligned} 1 &= c \int_0^{\theta_{\max}} \sin\theta \, d\theta \\ &= c(1 - \cos\theta_{\max}). \end{aligned}$$

So $p(\theta) = \sin\theta(1 - \cos\theta_{\max})$ and $p(\omega) = 1/(2\pi(1 - \cos\theta_{\max}))$.

INFINITY 1002
LightSample 710
Normal 65
Point 63
PointLight 610
PointLight::Intensity 611
PointLight::lightPos 611
Ray 66
Scene 22
Spectrum 263
SpotLight::Sample_L() 713
UniformSampleSphere() 664
UniformSpherePdf() 664
Vector 57

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float UniformConePdf(float cosThetaMax) {
    return 1.f / (2.f * M_PI * (1.f - cosThetaMax));
}
```

The PDF can be integrated to find the CDF, and the sampling technique,

$$\cos\theta = (1-\xi) + \xi\cos\theta_{\max},$$

follows. There are two `UniformSampleCone()` functions that implement this sampling technique: the first samples about the (0, 0, 1) axis, and the second (not shown here) takes three basis vectors for the coordinate system to be used where samples taken are with respect to the z axis of the given coordinate system.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector UniformSampleCone(float u1, float u2, float costhetamax) {
    float costheta = (1.f - u1) + u1 * costhetamax;
    float sintheta = sqrtf(1.f - costheta*costheta);
    float phi = u2 * 2.f * M_PI;
    return Vector(cosf(phi) * sintheta, sinf(phi) * sintheta, costheta);
}
```

The ray sampling method for the light uses these utility routines to choose a random ray inside the spotlight's cone.

⟨*SpotLight Method Definitions*⟩ +≡
```
Spectrum SpotLight::Sample_L(const Scene *scene, const LightSample &ls,
        float u1, float u2, float time, Ray *ray, Normal *Ns,
        float *pdf) const {
    Vector v = UniformSampleCone(ls.uPos[0], ls.uPos[1], cosTotalWidth);
    *ray = Ray(lightPos, LightToWorld(v), 0.f, INFINITY, time);
    *Ns = (Normal)ray->d;
    *pdf = UniformConePdf(cosTotalWidth);
    return Intensity * Falloff(ray->d);
}
```

GonioPhotometricLight 618
INFINITY 1002
Light::LightToWorld 606
LightSample 710
M_PI 1002
Normal 65
PointLight 610
ProjectionLight 614
ProjectionLight::
cosTotalWidth 617
Ray 66
Scene 22
Spectrum 263
SpotLight 612
SpotLight::Falloff() 614
SpotLight::Intensity 613
UniformConePdf() 713
UniformSampleCone() 713
Vector 57

Projection Lights and Goniophotometric Lights

The sampling routines for `ProjectionLights` and `GonioPhotometricLights` are essentially the same as `SpotLights` and `PointLights`, respectively. For sampling outgoing rays, `ProjectionLights` sample uniformly from the cone that encompasses their projected image map (hence the need to compute `ProjectionLight::cosTotalWidth` in the constructor), and those for `GonioPhotometricLights` sample uniformly over the unit sphere. An exercise at the end of this chapter discusses possible improvements to these sampling methods that better account for the directional variation of these lights.

Directional Lights

Sampling the incident illumination from a directional light at a given point is trivial. As with `PointLight` and its relatives, the `Sample_L()` method was implemented in Chapter 12 and its PDF method returns zero.

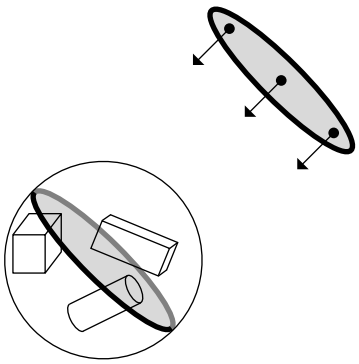


Figure 14.9: To sample an outgoing ray direction for a distant light source, the `DistantLight::Sample_L()` method finds the disk oriented in the light's direction that is large enough so that the entire scene can be intersected by rays leaving the disk in the light's direction. Ray origins are sampled uniformly by area on this disk, and ray directions are given directly by the light's direction.

Sampling a ray from the light's distribution of outgoing rays is a more interesting problem. The ray's direction is determined in advance by a delta distribution; it must be the same as the light's negated direction. For its origin, there are an infinite number of 3D points where it could start. How should we choose an appropriate one, and how do we compute its density?

The desired property is that rays intersect points in the scene that are illuminated by the distant light with uniform probability. One way to do this is to construct a disk that has the same radius as the scene's bounding sphere and has a normal that is oriented with the light's direction, and then choose a random point on this disk, using the `ConcentricSampleDisk()` function (Figure 14.9). Once this point has been chosen, if the point is displaced along the light's direction by the scene's bounding sphere radius and used as the origin of the light ray, the ray origin will be outside the bounding sphere of the scene, but will intersect it.

This is a valid sampling approach, since by construction it has nonzero probability of sampling all incident rays into the sphere due to the directional light. The area component of the sampling density is uniform and therefore equal to the reciprocal of the area of the disk that was sampled. The directional density is given by a delta distribution based on the light's direction; therefore, it isn't included in the returned PDF value.

⟨*DistantLight Method Definitions*⟩ +≡

```
Spectrum DistantLight::Sample_L(const Scene *scene,
        const LightSample &ls, float u1, float u2, float time,
        Ray *ray, Normal *Ns, float *pdf) const {
    ⟨Choose point on disk oriented toward infinite light direction 715⟩
    ⟨Set ray origin and direction for infinite light ray 715⟩
    *Ns = (Normal)ray->d;
```

ConcentricSampleDisk() 667
LightSample 710
M_PI 1002
Normal 65
Ray 66
Scene 22
Spectrum 263

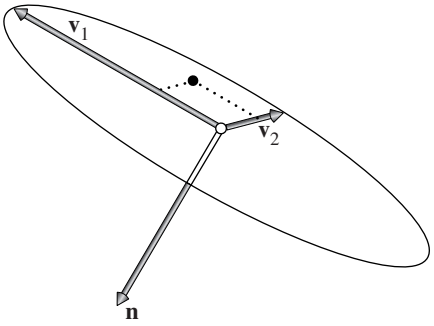


Figure 14.10: Given sample points (d_1, d_2) on the canonical unit disk, points on an arbitrarily oriented and sized disk with normal $\mathbf{n}$ can be found by computing an arbitrary coordinate system $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{n})$, and then computing points on the disk with the offset $d_1\mathbf{v}_1 + d_2\mathbf{v}_2$ from the disk's center.

```
    *pdf = 1.f / (M_PI * worldRadius * worldRadius);
    return L;
}
```

Choosing the point on the oriented disk is a simple application of vector algebra. We construct a coordinate system with two vectors perpendicular to the disk's normal (the light's direction); see Figure 14.10. Given a random point on the canonical unit disk, computing the offsets from the disk's center with respect to its coordinate vectors gives the corresponding point.

⟨*Choose point on disk oriented toward infinite light direction*⟩ ≡ 714

```
Point worldCenter;
float worldRadius;
scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
Vector v1, v2;
CoordinateSystem(lightDir, &v1, &v2);
float d1, d2;
ConcentricSampleDisk(ls.uPos[0], ls.uPos[1], &d1, &d2);
Point Pdisk = worldCenter + worldRadius * (d1 * v1 + d2 * v2);
```

And finally, the point is offset along the light direction and the ray is initialized:

⟨*Set ray origin and direction for infinite light ray*⟩ ≡ 714

```
*ray = Ray(Pdisk + worldRadius * lightDir, -lightDir, 0.f, INFINITY,
           time);
```

BBox::BoundingSphere() 74
ConcentricSampleDisk() 667
CoordinateSystem() 63
INFINITY 1002
Point 63
Ray 66
Scene::WorldBound() 24
Vector 57

14.6.3 AREA LIGHTS

Recall that area lights are defined by attaching an emission profile to a shape. Therefore, in order to sample incident illumination from such a light source, it is useful to be able to generate samples over the surface of the shape. To make this possible, we will add

sampling methods to the Shape class that sample random points on their surfaces.[2] The AreaLight sampling methods then call these methods.

Sampling Shapes

There are two shape sampling methods, both named Shape::Sample(). The first chooses points on the surface using some sampling distribution with respect to surface area on the shape and returns the position and surface normal of the point chosen.

⟨*Shape Interface*⟩ +≡ 108
```
virtual Point Sample(float u1, float u2, Normal *Ns) const {
    Severe("Unimplemented Shape::Sample() method called");
    return Point();
}
```

The Shapes that implement this method will almost always sample uniformly by area on their surface. Therefore, we will provide a default implementation of the Shape::Pdf() method corresponding to this sampling method that returns the corresponding PDF: the reciprocal of the surface area.

⟨*Shape Interface*⟩ +≡ 108
```
virtual float Pdf(const Point &Pshape) const {
    return 1.f / Area();
}
```

The second shape sampling method also takes the point from which the surface of the shape is being integrated over. This method is particularly useful for lighting, since the caller can pass in the point to be lit and allow shape implementations to ensure that they only sample the portion of the shape that is potentially visible from that point. The default implementation ignores the additional point and calls the earlier sampling method.

⟨*Shape Interface*⟩ +≡ 108
```
virtual Point Sample(const Point &P, float u1, float u2,
                     Normal *Ns) const {
    return Sample(u1, u2, Ns);
}
```

Unlike the first Shape sampling method, which generates points on the shape according to a probability density with respect to surface area on the shape, the second one uses a density with respect to solid angle from the point p. This difference stems from the fact that the area light sampling routines evaluate the direct lighting integral as an integral over directions from p—expressing these sampling densities with respect to solid angle at p is more convenient. Therefore, the standard implementation of the second Pdf()

AreaLight 623
Normal 65
Point 63
Sample 343
Severe() 1005
Shape 108
Shape::Area() 113
Shape::Pdf() 717
Shape::Sample() 716

2 Notice that the Shape::Sample() method is not a C++ "pure virtual" function. This is deliberate; not all shapes will implement a Sample() method. Some shapes are very difficult to sample (e.g., fractals), and it would be unreasonable to require those shapes to implement a Sample method just to be used as geometry in the system. Shapes like these should return false from their CanIntersect() method and should be able to refine themselves into shapes that do implement this method.

method here transforms the density from one defined over area to one defined over solid angle from p.

⟨*Shape Method Definitions*⟩ +≡
```
float Shape::Pdf(const Point &p, const Vector &wi) const {
    ⟨Intersect sample ray with area light geometry 717⟩
    ⟨Convert light sample weight to solid angle measure 717⟩
    return pdf;
}
```

Given a point p and direction ω_i, the `Pdf()` method determines if the ray (p, ω_i) intersects the shape. If the ray doesn't intersect the shape at all, the probability that the shape would have chosen the direction ω_i can be assumed to be zero (a well-written sampling algorithm wouldn't generate such a sample, and in any case the light's contribution will be zero for such directions, so using a zero probability density is fine).

Note that this ray intersection test is only between the ray and the single shape that is the area light source. The rest of the scene geometry is ignored, thus the intersection test is reasonably efficient.

⟨*Intersect sample ray with area light geometry*⟩ ≡ 717
```
DifferentialGeometry dgLight;
Ray ray(p, wi, 1e-3f);
float thit, rayEpsilon;
if (!Intersect(ray, &thit, &rayEpsilon, &dgLight)) return 0.;
```

To compute the value of the PDF with respect to solid angle from p, this method starts by computing the PDF with respect to surface area. Conversion from a density with respect to area to a density with respect to solid angle requires division by the factor

$$\frac{d\omega_i}{dA} = \frac{\cos\theta_o}{r^2},$$

where θ_o is the angle between the ray from the point on the light to the receiving point p and the light's surface normal, and r^2 is the distance between the point on the light and the point being shaded (recall Section 5.5).

⟨*Convert light sample weight to solid angle measure*⟩ ≡ 717
```
float pdf = DistanceSquared(p, ray(thit)) /
            (AbsDot(dgLight.nn, -wi) * Area());
```

AbsDot() 61
DifferentialGeometry 102
DiffuseAreaLight 625
DiffuseAreaLight::Sample_L() 718
DistanceSquared() 65
Point 63
Ray 66
Shape 108
Shape::Area() 113
Shape::Intersect() 111
Vector 57

Area Light Sampling Methods

Given shape sampling methods, the `DiffuseAreaLight::Sample_L()` methods are quite straightforward. Most of the hard work is done by the `Shapes`, and the `DiffuseAreaLight` mostly just needs to compute emitted radiance values. Recall that the shape or shapes that define the light are stored in a `ShapeSet` that handles sampling from collections of shapes. Its sampling routines will be defined in Section 14.6.4.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
Spectrum DiffuseAreaLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    Normal ns;
    Point ps = shapeSet->Sample(p, ls, &ns);
    *wi = Normalize(ps - p);
    *pdf = shapeSet->Pdf(p, *wi);
    visibility->SetSegment(p, pEpsilon, ps, 1e-3f, time);
    Spectrum Ls = L(ps, ns, -*wi);
    return Ls;
}
```

The variant of `Shape::Pdf()` called here is the one that returns a density with respect to solid angle, so the value from the light's `Pdf()` method can be returned directly.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
float DiffuseAreaLight::Pdf(const Point &p, const Vector &wi) const {
    return shapeSet->Pdf(p, wi);
}
```

The method for sampling a ray leaving an area light is also easily implemented in terms of the shape sampling methods. The first variant of `Shape::Sample()` is used to find the ray origin, sampled from some density over the surface. Because the `DiffuseAreaLight` emits uniform radiance in all directions, a uniform distribution of directions is used for the ray direction. Because radiance is only emitted from the side of the light the surface normal lies on, if the sampled direction is in the opposite hemisphere than the surface normal, it is flipped to lie in the same hemisphere so that samples aren't wasted in directions where there is no emission.

The PDF for sampling the ray is the product of the PDF for sampling its origin with respect to area on the surface with the PDF for sampling a uniform direction on the hemisphere, $1/(2\pi)$.

⟨*DiffuseAreaLight Method Definitions*⟩ +≡
```
Spectrum DiffuseAreaLight::Sample_L(const Scene *scene,
        const LightSample &ls, float u1, float u2, float time,
        Ray *ray, Normal *Ns, float *pdf) const {
    Point org = shapeSet->Sample(ls, Ns);
    Vector dir = UniformSampleSphere(u1, u2);
    if (Dot(dir, *Ns) < 0.) dir *= -1.f;
    *ray = Ray(org, dir, 1e-3f, INFINITY, time);
    *pdf = shapeSet->Pdf(org) * INV_TWOPI;
    Spectrum Ls = L(org, *Ns, dir);
    return Ls;
}
```

Sampling Disks

The `Disk` sampling method uses the concentric disk sampling function to find a point on the unit disk and then scales and offsets this point to lie on the disk of a given

DiffuseAreaLight 625
DiffuseAreaLight::L() 626
Disk 129
Dot() 60
INFINITY 1002
INV_TWOPI 1002
LightSample 710
Normal 65
Point 63
Ray 66
Scene 22
Shape::Pdf() 717
Shape::Sample() 716
ShapeSet::Pdf() 724
ShapeSet::Sample() 723
Spectrum 263
UniformSampleSphere() 664
Vector 57
Vector::Normalize() 63
VisibilityTester 608
VisibilityTester::
SetSegment() 609

radius and height. Note that this method does not account for partial disks due to `Disk::innerRadius` being nonzero or `Disk::phiMax` being less than 2π. Solutions to this bug are discussed in an exercise at the end of the chapter.

⟨*Disk Method Definitions*⟩ +≡
```
Point Disk::Sample(float u1, float u2, Normal *Ns) const {
    Point p;
    ConcentricSampleDisk(u1, u2, &p.x, &p.y);
    p.x *= radius;
    p.y *= radius;
    p.z = height;
    *Ns = Normalize((*ObjectToWorld)(Normal(0,0,1)));
    if (ReverseOrientation) *Ns *= -1.f;
    return (*ObjectToWorld)(p);
}
```

Sampling Cylinders

Uniform sampling on cylinders is straightforward. The height and ϕ value are sampled uniformly. Intuitively, it can be understood that this works because a cylinder is just a rolled-up rectangle.

⟨*Cylinder Method Definitions*⟩ +≡
```
Point Cylinder::Sample(float u1, float u2, Normal *Ns) const {
    float z = Lerp(u1, zmin, zmax);
    float t = u2 * phiMax;
    Point p = Point(radius * cosf(t), radius * sinf(t), z);
    *Ns = Normalize((*ObjectToWorld)(Normal(p.x, p.y, 0.)));
    if (ReverseOrientation) *Ns *= -1.f;
    return (*ObjectToWorld)(p);
}
```

ConcentricSampleDisk() 667
Cross() 62
Cylinder::phiMax 126
Cylinder::radius 126
Cylinder::zmax 126
Cylinder::zmin 126
Disk::height 130
Disk::innerRadius 130
Disk::phiMax 130
Disk::radius 130
Lerp() 1000
Normal 65
Point 63
Sample 343
Shape::ObjectToWorld 108
Shape::
ReverseOrientation 108
Triangle 139
UniformSampleTriangle() 671
Vector::Normalize() 63

Sampling Triangles

The `UniformSampleTriangle()` function, defined in the previous chapter, returns the barycentric coordinates for a uniformly sampled point on a triangle. The point on a particular triangle for those barycentrics is easily computed.

⟨*TriangleMesh Method Definitions*⟩ +≡
```
Point Triangle::Sample(float u1, float u2, Normal *Ns) const {
    float b1, b2;
    UniformSampleTriangle(u1, u2, &b1, &b2);
    ⟨Get triangle vertices in p1, p2, and p3 140⟩
    Point p = b1 * p1 + b2 * p2 + (1.f - b1 - b2) * p3;
    Normal n = Normal(Cross(p2-p1, p3-p1));
    *Ns = Normalize(n);
    if (ReverseOrientation) *Ns *= -1.f;
    return p;
}
```

Sampling Spheres

As with `Disks`, the sampling method here does not handle partial spheres; an exercise at the end of the chapter discusses this issue further. If the sampling method is not given an external point that's being lit, sampling a point on a sphere is extremely simple. It just uses the `UniformSampleSphere()` function to generate a point on the unit sphere and scales the point by the sphere's radius.

⟨*Sphere Method Definitions*⟩ +≡

```
Point Sphere::Sample(float u1, float u2, Normal *ns) const {
    Point p = Point(0,0,0) + radius * UniformSampleSphere(u1, u2);
    *ns = Normalize((*ObjectToWorld)(Normal(p.x, p.y, p.z)));
    if (ReverseOrientation) *ns *= -1.f;
    return (*ObjectToWorld)(p);
}
```

For the sphere sampling method that is given a point being illuminated, however, it can do better. While uniform sampling over its surface leads to a correct estimate, a better approach is not to sample points on the sphere that are definitely not visible to the point being shaded (such as those on the back side of the sphere as seen from the point). The sampling routine will instead uniformly sample directions over the solid angle subtended by the sphere from the point being shaded. It generates directions inside this cone of directions by sampling an offset θ from the center vector ω_c and then sampling a rotation angle ϕ around the vector.

As seen from the point being shaded p the sphere subtends an angle of

$$\theta_{\max} = \arcsin\left(\frac{r}{|\mathrm{p} - \mathrm{p_c}|}\right) = \arccos\sqrt{1 - \left(\frac{r}{|\mathrm{p} - \mathrm{p_c}|}\right)^2}, \qquad [14.2]$$

where r is the radius of the sphere and $\mathrm{p_c}$ is its center (Figure 14.11). The implementation uniformly samples directions inside this cone and computes their intersection with the sphere to get the sample position on the light source.

⟨*Sphere Method Definitions*⟩ +≡

```
Point Sphere::Sample(const Point &p, float u1, float u2,
                     Normal *ns) const {
    ⟨Compute coordinate system for sphere sampling 721⟩
    ⟨Sample uniformly on sphere if p is inside it 721⟩
    ⟨Sample sphere uniformly inside subtended cone 722⟩
}
```

This process is most easily done if we first compute a coordinate system to use for sampling the sphere where the z axis is the vector between the sphere's center and the point being illuminated:

Disk 129
Normal 65
Point 63
Sample 343
Shape::ObjectToWorld 108
Shape::
ReverseOrientation 108
Sphere::radius 116
UniformSampleSphere() 664
Vector::Normalize() 63

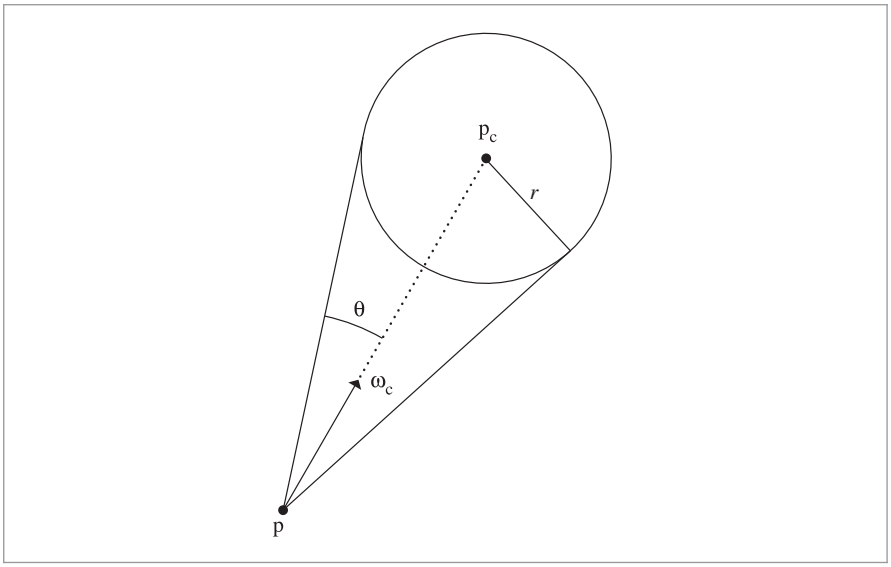


Figure 14.11: To sample points on a spherical light source, we can uniformly sample within the cone of directions around a central vector ω_c with an angular spread of up to θ. Trigonometry can be used to derive the value of $\sin\theta$, $r/|p_c - p|$.

⟨*Compute coordinate system for sphere sampling*⟩ ≡ 720
```
Point Pcenter = (*ObjectToWorld)(Point(0,0,0));
Vector wc = Normalize(Pcenter - p);
Vector wcX, wcY;
CoordinateSystem(wc, &wcX, &wcY);
```

It is important to be careful about points that lie inside the sphere. If this happens, the entire sphere should be sampled, since the whole sphere is clearly visible from inside it. Notice that we use a small constant, `1e-4`, to do this check; this makes the code more robust when points are very close to the sphere.

⟨*Sample uniformly on sphere if* p *is inside it*⟩ ≡ 720
```
if (DistanceSquared(p, Pcenter) - radius*radius < 1e-4f)
    return Sample(u1, u2, ns);
```

If the point being lit is outside the sphere, this method computes the cosine of the subtended angle $\theta_{\max}$ using Equation (14.2) and then generates a random ray inside the subtended cone using the `UniformSampleCone()` function and intersects it with the sphere to get the sample point on its surface.

Note that we must be careful about precision errors here. If the generated ray just grazes the edge of the sphere, the `Sphere::Intersect()` routine might unexpectedly return `false`. In this case, the implementation sets the `thit` value so that it corresponds to the point where the ray should have intersected the sphere at the edge.

CoordinateSystem() 63
DistanceSquared() 65
Point 63
Shape::ObjectToWorld 108
Sphere::Intersect() 117
Sphere::radius 116
Sphere::Sample() 720
UniformSampleCone() 713
Vector 57
Vector::Normalize() 63

⟨*Sample sphere uniformly inside subtended cone*⟩ ≡ 720
```
float sinThetaMax2 = radius*radius / DistanceSquared(p, Pcenter);
float cosThetaMax = sqrtf(max(0.f, 1.f - sinThetaMax2));
DifferentialGeometry dgSphere;
float thit, rayEpsilon;
Point ps;
Ray r(p, UniformSampleCone(u1, u2, cosThetaMax, wcX, wcY, wc), 1e-3f);
if (!Intersect(r, &thit, &rayEpsilon, &dgSphere))
    thit = Dot(Pcenter - p, Normalize(r.d));
ps = r(thit);
*ns = Normal(Normalize(ps - Pcenter));
if (ReverseOrientation) *ns *= -1.f;
return ps;
```

⟨*Sphere Method Definitions*⟩ +≡
```
float Sphere::Pdf(const Point &p, const Vector &wi) const {
    Point Pcenter = (*ObjectToWorld)(Point(0,0,0));
    ⟨Return uniform weight if point inside sphere 722⟩
    ⟨Compute general sphere weight 722⟩
}
```

To compute the PDF for sampling a direction towards a sphere from a given point, we must first determine which of the two sampling strategies for points inside and outside the sphere would be used for the point. If the shading point was inside the sphere, a uniform sampling strategy was used, in which case, the implementation hands off the `Pdf()` call to the parent class, which takes care of the solid angle conversion.

⟨*Return uniform weight if point inside sphere*⟩ ≡ 722
```
if (DistanceSquared(p, Pcenter) - radius*radius < 1e-4f)
    return Shape::Pdf(p, wi);
```

In the general case, we simply recompute the angle subtended by the sphere and call `UniformConePdf()`. Note that no conversion of sampling measures is required here because `UniformConePdf()` already returns weights with respect to a solid angle measure.

⟨*Compute general sphere weight*⟩ ≡ 722
```
float sinThetaMax2 = radius*radius / DistanceSquared(p, Pcenter);
float cosThetaMax = sqrtf(max(0.f, 1.f - sinThetaMax2));
return UniformConePdf(cosThetaMax);
```

14.6.4 SHAPESET SAMPLING

A `ShapeSet` is allocated in the `DiffuseAreaLight` constructor to store the shape or shapes that define the light. It conforms to the `Shape` interface so that the light source can just store a single `Shape` pointer, regardless of how many shapes are in the light source. The `ShapeSet` thus must implement the `Shape`'s sampling routines.

In order to be able to uniformly sample the overall shape by surface area, we would like to sample individual shapes in the collection with probability based on the ratio of their area to the total area of all of the shapes. Therefore, the `ShapeSet` constructor uses the

DifferentialGeometry 102
DiffuseAreaLight 625
DistanceSquared() 65
Normal 65
Point 63
Ray 66
Shape 108
Shape::ObjectToWorld 108
Shape::Pdf() 717
Shape::
ReverseOrientation 108
ShapeSet 626
Sphere::Intersect() 117
Sphere::radius 116
UniformConePdf() 713
UniformSampleCone() 713
Vector 57
Vector::Normalize() 63

Distribution1D to store a discrete CDF for sampling each shape according to individual probabilities given by this ratio.

⟨*Compute total area of shapes in* ShapeSet *and area CDF*⟩ ≡
```
sumArea = 0.f;
for (uint32_t i = 0; i < shapes.size(); ++i) {
    float a = shapes[i]->Area();
    areas.push_back(a);
    sumArea += a;
}
areaDistribution = new Distribution1D(&areas[0], areas.size());
```

⟨*ShapeSet Private Data*⟩ +≡ 626
```
float sumArea;
vector<float> areas;
Distribution1D *areaDistribution;
```

Here we'll show the variants of the ShapeSet's Sample() and Pdf() methods that take the point being shaded and sample by solid angle. The variants for the case of sampling with respect to area and sampling outgoing rays are similar and so are elided. For Sample(), a shape is selected according to its area and its Sample() method is called.

⟨*ShapeSet Method Definitions*⟩ ≡
```
Point ShapeSet::Sample(const Point &p, const LightSample &ls,
                       Normal *Ns) const {
    int sn = areaDistribution->SampleDiscrete(ls.uComponent, NULL);
    Point pt = shapes[sn]->Sample(p, ls.uPos[0], ls.uPos[1], Ns);
    ⟨Find closest intersection of ray with shapes in ShapeSet 723⟩
}
```

It may be that one of the other shapes is visible along the ray from the point being illuminated to the sampled point; in this case, it's important that we return the point on that shape rather than the originally sampled point. The implementation here checks for intersection with all of the shapes; this may be inefficient for large numbers of shapes.

⟨*Find closest intersection of ray with shapes in* ShapeSet⟩ ≡ 723
```
Ray r(p, pt-p, 1e-3f, INFINITY);
float rayEps, thit = 1.f;
bool anyHit = false;
DifferentialGeometry dg;
for (uint32_t i = 0; i < shapes.size(); ++i)
    anyHit |= shapes[i]->Intersect(r, &thit, &rayEps, &dg);
if (anyHit) *Ns = dg.nn;
return r(thit);
```

To compute the value of the PDF, we just need to iterate over all of the Shapes and call their Pdf() methods, taking the area-weighted average of their PDFs since shapes are selected by area in ShapeSet::Sample(). Executing this method will be inefficient when large numbers of lights are in the ShapeSet, due to the expense of evaluating the PDF for each of them (recall that a ray–shape intersection is performed in the Shape::Pdf()

DifferentialGeometry 102
DifferentialGeometry::nn 102
Distribution1D 648
Distribution1D::SampleDiscrete() 650
INFINITY 1002
LightSample 710
LightSample::uComponent 710
LightSample::uPos 710
Normal 65
Point 63
Ray 66
Sample 343
Shape 108
Shape::Area() 113
Shape::Intersect() 111
Shape::Sample() 716
ShapeSet 626
ShapeSet::areaDistribution 723
ShapeSet::areas 723
ShapeSet::Sample() 723
ShapeSet::shapes 626
ShapeSet::sumArea 723

methods). The ShapeSet could therefore maintain a small acceleration structure here so that it could avoid unnecessarily calling this method for the Shapes that the ray $(\mathrm{p}, \omega_\mathrm{i})$ doesn't intersect at all.

⟨*ShapeSet Method Definitions*⟩ +≡
```
float ShapeSet::Pdf(const Point &p, const Vector &wi) const {
    float pdf = 0.f;
    for (uint32_t i = 0; i < shapes.size(); ++i)
        pdf += areas[i] * shapes[i]->Pdf(p, wi);
    return pdf / sumArea;
}
```

14.6.5 INFINITE AREA LIGHTS

The InfiniteAreaLight, defined in Section 12.5, can be considered to be an infinitely large sphere that surrounds the entire scene, illuminating it from all directions. The environment maps used with InfiniteAreaLights often have substantial variation along different directions: consider, for example, an environment map of the sky during daytime, where the relatively small number of directions that the sun subtends will be thousands of times brighter than the rest of the directions. Given this substantial variation, implementing a sampling method for InfiniteAreaLights that matches the illumination distribution substantially reduces variance in images.

Figure 14.12 shows two images of the Audi TT car model illuminated by the morning skylight environment map from Figure 12.16. The top image was rendered using a simple cosine-weighted sampling distribution for selecting incident illumination directions, while the bottom image was rendered using the sampling method implemented in this section. Both images used just 32 shadow samples per pixel. For the same number of samples taken and with negligible additional computational cost, this sampling method computes a substantially better result with much lower variance.

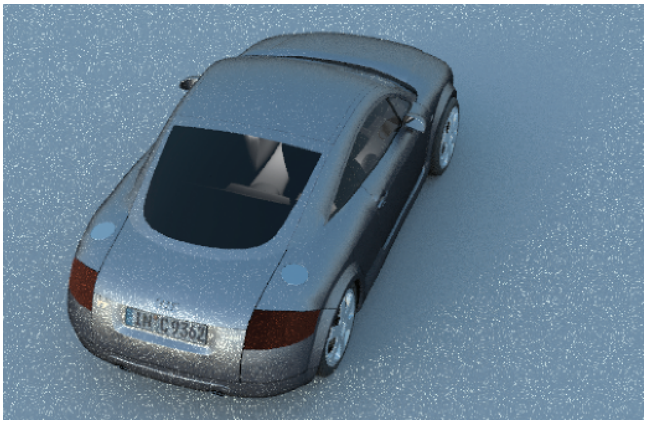
There are three main steps to the sampling approach implemented here:

1. We define a piecewise-constant 2D probability distribution function in (u, v) image coordinates $p(u, v)$ that corresponds to the distribution of the radiance represented by the environment map.
2. We apply the sampling method from Section 13.6.5 to transform uniformly distributed 2D random numbers to samples drawn from the piecewise-constant $p(u, v)$ distribution.
3. We define a probability density function over directions on the unit sphere based on the probability density over (u, v).

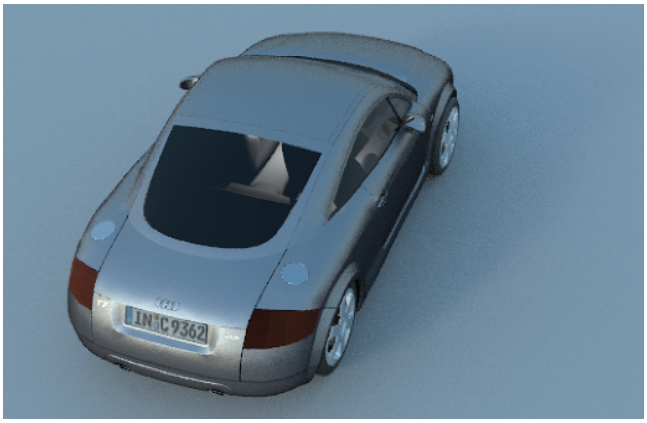
The combination of these three steps makes it possible to generate samples on the sphere of directions according to a distribution that matches the radiance function very closely, leading to substantial variance reduction.

We will start by defining the fragment ⟨*Initialize sampling PDFs for infinite area light*⟩ at the end of the InfiniteAreaLight constructor.

InfiniteAreaLight 629
Point 63
Shape 108
Shape::Pdf() 717
ShapeSet::areas 723
ShapeSet::shapes 626
ShapeSet::sumArea 723
Vector 57



(a)



(b)

Figure 14.12: TT Car Model Illuminated by the Morning Skylight Environment Map. These images were rendered with four image samples per pixel and eight light source samples per image sample. (a) The result of using a uniform sampling distribution, and (b) the improvement from the importance sampling method implemented here. A total of just 32 light samples per pixel gives an excellent result with this approach.

⟨*Initialize sampling PDFs for infinite area light*⟩ ≡ 630
```
⟨Compute scalar-valued image img from environment map 727⟩
⟨Compute sampling distributions for rows and columns of image 727⟩
delete[] img;
```

Spectrum::y() 273

The first step is to transform the continuously defined spectral radiance function defined by the environment map's texels to a piecewise-constant scalar function by computing its luminance at a set of sample points using the `Spectrum::y()` method. There are three things to note in the code below that does this computation.

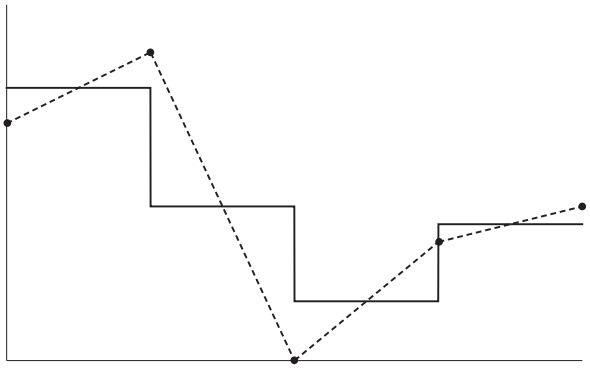


Figure 14.13: Finding a Piecewise-Constant Function (solid lines) that Approximates a Piecewise Linear Function (dashed lines) for Use as a Sampling Distribution. Even though some of the sample points that define the piecewise linear function (solid dots) may be zero-valued, the piecewise-constant sampling function must not be zero over any range where the actual function is nonzero. A reasonable approach to avoid this case, shown here and implemented in the `InfiniteAreaLight` sampling routines, is to find the average value of the function over some range and use that to define the piecewise-constant function.

First, it computes values of the radiance function at the same number of points as there are texels in the original image map. It could use either more or fewer points, leading to a corresponding increase or decrease in memory use while still generating a valid sampling distribution, however. These values work well, though, as fewer points would lead to a sampling distribution that didn't match the function as well while more would mostly waste memory with little incremental benefit.

The second thing of note in this code is that the piecewise constant function values being stored here in `img` are found by slightly blurring the radiance function with the `MIPMap::Lookup()` method (rather than just copying the corresponding texel values). The motivation for this is subtle; Figure 14.13 illustrates the idea in 1D. Because the continuous radiance function used for rendering is reconstructed by bilinearly interpolating between texels in the image, just because some texel is completely black, for example, the radiance function may be nonzero a tiny distance away from it due to a neighboring texel's contribution. Because we are using a piecewise-constant function for sampling rather than a piecewise-linear one, it must account for this issue in order to ensure greater-than-zero probability of sampling any point where the radiance function is nonzero.[3]

Finally, each image value in the `img` buffer is multiplied by the value of $\sin\theta$ corresponding to the θ value each row has when the latitude–longitude image is mapped to the sphere. Note that this multiplication has no effect on the sampling method's correctness:

3 Alternatively, we could use a piecewise-linear function for importance sampling and thus match the radiance function exactly. However, it's easier to draw samples from a piecewise-constant function's distribution, and, because environment maps generally have a large number of texel samples, a piecewise-constant function suffices to match its distribution well.

because the value of $\sin\theta$ is always greater than zero over the $[0, \pi]$ range, we are just reshaping the sampling PDF. The motivation for adjusting the PDF is to eliminate the effect of the distortion from mapping the 2D image to the unit sphere in the sampling method here; the details will be fully explained later in this section.

⟨*Compute scalar-valued image* `img` *from environment map*⟩ ≡ 725
```
float filter = 1.f / max(width, height);
float *img = new float[width*height];
for (int v = 0; v < height; ++v) {
    float vp = (float)v / (float)height;
    float sinTheta = sinf(M_PI * float(v+.5f)/float(height));
    for (int u = 0; u < width; ++u) {
        float up = (float)u / (float)width;
        img[u+v*width] = radianceMap->Lookup(up, vp, filter).y();
        img[u+v*width] *= sinTheta;
    }
}
```

Given this filtered and scaled image, the `Distribution2D` structure handles computing and storing the 2D PDF.

⟨*Compute sampling distributions for rows and columns of image*⟩ ≡ 725
```
distribution = new Distribution2D(img, width, height);
```

⟨*InfiniteAreaLight Private Data*⟩ +≡ 629
```
Distribution2D *distribution;
```

Given this precomputed data, the task of the sampling method is relatively straightforward. Given a pair of uniformly distributed random variables (ξ_1, ξ_2) over $[0, 1]^2$, it draws a sample from the function's distribution using the sampling algorithm described in Section 13.6.5, giving a (u, v) value and the value of the PDF for taking this sample, $p(u, v)$.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Sample_L(const Point &p, float pEpsilon,
        const LightSample &ls, float time, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    ⟨Find (u, v) sample coordinates in infinite light texture 727⟩
    ⟨Convert infinite light sample point to direction 728⟩
    ⟨Compute PDF for sampled infinite light direction 728⟩
    ⟨Return radiance value for infinite light direction 729⟩
}
```

⟨*Find* (u, v) *sample coordinates in infinite light texture*⟩ ≡ 727
```
float uv[2], mapPdf;
distribution->SampleContinuous(ls.uPos[0], ls.uPos[1], uv, &mapPdf);
```

The (u, v) sample is mapped to spherical coordinates by

$$(\theta, \phi) = (\pi v, 2\pi u) ,$$

and then the spherical coordinates formula gives the direction $\omega = (x, y, z)$.

Distribution2D 672
Distribution2D::
SampleContinuous() 673
InfiniteAreaLight::
distribution 727
LightSample 710
LightSample::uPos 710
MIPMap::Lookup() 540
M_PI 1002
Point 63
Spectrum 263
Spectrum::y() 273
Vector 57
VisibilityTester 608

⟨*Convert infinite light sample point to direction*⟩ ≡ 727

```
float theta = uv[1] * M_PI, phi = uv[0] * 2.f * M_PI;
float costheta = cosf(theta), sintheta = sinf(theta);
float sinphi = sinf(phi), cosphi = cosf(phi);
*wi = LightToWorld(Vector(sintheta * cosphi, sintheta * sinphi,
                          costheta));
```

Recall that the probability density values returned by the light source sampling routines must be defined in terms of the solid angle measure on the unit sphere. Therefore, this routine must now compute the transformation between the sampling density used, which was the image function over $[0, 1]^2$, and the corresponding density after the image has been mapped to the unit sphere with the latitude–longitude mapping. (Recall that the latitude–longitude parameterization of an image (θ, ϕ) is $x = r \sin\theta \cos\phi$, $y = r \sin\theta \sin\phi$, and $z = r \cos\theta$.)

First, consider the function g that maps from (u, v) to (θ, ϕ),

$$g(u, v) = (\pi v, 2\pi u) .$$

The absolute value of the determinant of the Jacobian $|J_g|$ is $2\pi^2$. Applying the multidimensional change of variables equation from Section 13.5.1, we can find the density in terms of spherical coordinates (θ, ϕ).

$$p(\theta, \phi) = \frac{p(u, v)}{2\pi^2}.$$

From the definition of spherical coordinates, it is easy to determine that the absolute value of the Jacobian for the mapping from (r, θ, ϕ) to (x, y, z) is $r^2 \sin\theta$. Since we are interested in the unit sphere, $r = 1$, and again applying the multidimensional change of variables equation to find the final relationship between probability densities,

$$p(\omega) = \frac{p(\theta, \phi)}{\sin\theta} = \frac{p(u, v)}{2\pi^2 \sin\theta}.$$

This is the key relationship for applying this technique: it lets us sample from the piecewise-constant distribution defined by the image map and transform the sample and its probability density to be in terms of directions on the unit sphere.

We can now see why the initialization routines multiplied the values of the piecewise-constant sampling function by a $\sin\theta$ term. Consider, for example, a constant-valued environment map: with the $p(u, v)$ sampling technique, all (θ, ϕ) values are equally likely to be chosen. Due to the mapping to directions on the sphere, however, this would lead to more directions being sampled near the poles of the sphere, *not* a uniform sampling of directions on the sphere, which would be a more desirable result. The $1/\sin\theta$ term in the $p(\omega)$ PDF corrects for this non-uniform sampling of directions so that correct results are computed in Monte Carlo estimates. Given this state of affairs, however, it's better to have modified the $p(u, v)$ sampling distribution so that it's less likely to select directions near the poles in the first place.

⟨*Compute PDF for sampled infinite light direction*⟩ ≡ 727

```
*pdf = mapPdf / (2.f * M_PI * M_PI * sintheta);
if (sintheta == 0.f) *pdf = 0.f;
```

Light::LightToWorld 606
M_PI 1002
Vector 57

We can finally initialize the VisibilityTester and return the radiance value for the chosen direction.

⟨*Return radiance value for infinite light direction*⟩ ≡ 727
```
visibility->SetRay(p, pEpsilon, *wi, time);
Spectrum Ls = Spectrum(radianceMap->Lookup(uv[0], uv[1]),
                       SPECTRUM_ILLUMINANT);
return Ls;
```

The InfiniteAreaLight::Pdf() method needs to convert the direction ω to the corresponding (u, v) coordinates in the sampling distribution. Given these, the PDF $p(u, v)$ is computed as the product of the two 1D PDFs by the Distribution2D::Pdf() method, which is adjusted here for mapping to the sphere as was done in the Sample_L() method.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
float InfiniteAreaLight::Pdf(const Point &, const Vector &w) const {
    Vector wi = WorldToLight(w);
    float theta = SphericalTheta(wi), phi = SphericalPhi(wi);
    float sintheta = sinf(theta);
    if (sintheta == 0.f) return 0.f;
    float p = distribution->Pdf(phi * INV_TWOPI, theta * INV_PI) /
           (2.f * M_PI * M_PI * sintheta);
    return p;
}
```

DistantLight 621
Distribution2D::Pdf() 673
InfiniteAreaLight 629
InfiniteAreaLight::Pdf() 729
INV_PI 1002
INV_TWOPI 1002
Light::WorldToLight 606
LightSample 710
MIPMap::Lookup() 540
M_PI 1002
Normal 65
Point 63
Ray 66
Scene 22
Spectrum 263
SPECTRUM_ILLUMINANT 277
SphericalPhi() 292
SphericalTheta() 292
Vector 57
VisibilityTester 608
VisibilityTester::
SetRay() 609

Generating a random ray leaving an infinite light source can be done by sampling a direction with the same approach as is used for direct lighting in the Sample_L() method above. The sampled ray's origin is then set using the same approach as was used for DistantLights in Section 14.6.2, where a disk that covers the scene's bounding sphere is oriented along the ray's direction. (Recall Figure 14.9.) We therefore won't include the ⟨*Compute direction for infinite light sample ray*⟩ or ⟨*Compute origin for infinite light sample ray*⟩ fragments here.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
Spectrum InfiniteAreaLight::Sample_L(const Scene *scene,
        const LightSample &ls, float u1, float u2, float time,
        Ray *ray, Normal *Ns, float *pdf) const {
    ⟨Compute direction for infinite light sample ray⟩
    ⟨Compute origin for infinite light sample ray⟩
    ⟨Compute InfiniteAreaLight ray PDF 730⟩
    Spectrum Ls = (radianceMap->Lookup(uv[0], uv[1]), SPECTRUM_ILLUMINANT);
    return Ls;
}
```

The PDF for these rays is just the product of the PDF for sampling the direction and the PDF for sampling a point on the disk.

⟨*Compute* InfiniteAreaLight *ray PDF*⟩ ≡ 729

```
float directionPdf = mapPdf / (2.f * M_PI * M_PI * sintheta);
float areaPdf = 1.f / (M_PI * worldRadius * worldRadius);
*pdf = directionPdf * areaPdf;
if (sintheta == 0.f) *pdf = 0.f;
```

* 14.7 VOLUME SCATTERING

The equation of transfer is the integral equation that describes the effect of participating media on the distribution of radiance in an environment. It is introduced in Section 16.1, and the VolumeIntegrators in Chapter 16 use Monte Carlo integration to compute estimates of (simplified) instances of this integral. The equation of transfer is an interesting example case for Monte Carlo integration since it can be an integral over an infinite range, of the general form

$$\int_0^\infty \mathrm{e}^{-g(t)t} f(t)\, \mathrm{d}t.$$

Thanks to the exponential term, this integral has a finite value in volume scattering applications.

It's not possible to sample from a uniform distribution to compute estimates of this integral. There's no valid uniform PDF over the range $[0, \infty)$, so some other sampling distribution must be used instead. In some cases, the function $g(t)$ is known to be constant and a natural sampling distribution is the one based on the exponential term; Section 13.3.1 derived the general sampling process. Thus, if samples T_i are drawn from this distribution, estimates of the integral can be computed by

$$\frac{1}{N}\sum_i^N \frac{\mathrm{e}^{-cT_i} f(T_i)}{c\,\mathrm{e}^{-cT_i}} = \frac{1}{N}\sum_i^N \frac{f(T_i)}{c}.$$

More generally, if the function $g(t)$ is not constant, a value of c for the exponential sampling distribution can be chosen arbitrarily (e.g., based on the average value of $g(t)$, if known), and the estimate is

$$\frac{1}{N}\sum_i^N \frac{\mathrm{e}^{-g(T_i)T_i} f(T_i)}{c\,\mathrm{e}^{-cT_i}}.$$

Because the extents of VolumeRegions in pbrt must be finite, the system never actually needs to estimate values of these integrals over infinite ranges, and a uniform sampling distribution can still be used. However, as the extents of VolumeRegions increase and as the $g(t)$ term becomes larger, uniform sampling is an increasingly inefficient sampling method as samples are taken in areas where the exponential function is very small, and the variance from not using importance sampling based on the exponential term increases. The implementations of VolumeIntegrators in Chapter 16 use uniform sampling, although Exercises 16.3, 16.4, and 16.5 at the end of that chapter discuss how to address this issue.

M_PI 1002
VolumeIntegrator 876
VolumeRegion 587

14.7.1 SAMPLING PHASE FUNCTIONS

It is useful to be able to draw samples from the distribution described by the Henyey–Greenstein phase functions for a number of applications, including applying importance sampling to computing direct lighting in participating media as well as for more general algorithms that compute the effect of multiple scattering in participating media. The PDF for this distribution is separable into θ and ϕ components, with $p(\phi) = 1/(2\pi)$ as usual. The distribution for θ can be shown to be

$$\cos\theta = \frac{1}{2g}\left(1 + g^2 - \left(\frac{1-g^2}{1-g+2g\xi}\right)^2\right)$$

if $g \neq 0$; otherwise, $\cos\theta = 1 - 2\xi$.

This sampling technique is implemented here in a routine that takes one vector ω, the asymmetry parameter g, and two random numbers; samples from this distribution; and uses the result to construct the corresponding outgoing ray.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
Vector SampleHG(const Vector &w, float g, float u1, float u2) {
    float costheta;
    if (fabsf(g) < 1e-3)
        costheta = 1.f - 2.f * u1;
    else {
        float sqrTerm = (1.f - g * g) /
                        (1.f - g + 2.f * g * u1);
        costheta = (1.f + g * g - sqrTerm * sqrTerm) / (2.f * g);
    }
    float sintheta = sqrtf(max(0.f, 1.f-costheta*costheta));
    float phi = 2.f * M_PI * u2;
    Vector v1, v2;
    CoordinateSystem(w, &v1, &v2);
    return SphericalDirection(sintheta, costheta, phi, v1, v2, w);
}
```

Because phase functions are already distribution functions, the probability density for sampling any particular direction is the value of the phase function for the given pair of directions.

⟨*Monte Carlo Function Definitions*⟩ +≡
```
float HGPdf(const Vector &w, const Vector &wp, float g) {
    return PhaseHG(w, wp, g);
}
```

CoordinateSystem() 63
M_PI 1002
PhaseHG() 585
SphericalDirection() 292
Vector 57

14.7.2 COMPUTING OPTICAL THICKNESS

Section 11.1.3 introduced optical thickness τ, which gives a measure of the total density of participating media that a ray passes through. For media with constant scattering

properties, optical thickness can be computed in closed form using Beer's law, Equation (11.1), although that chapter deferred the implementation of a method to compute τ for general media until after Monte Carlo integration had been introduced. Here, we will implement the general `DensityRegion::tau()` method, which computes an estimate of the optical thickness for a ray passing through an arbitrary medium.

Recall the definition of τ:

$$\tau(\mathrm{p} \to \mathrm{p}') = \int_0^d \sigma_\mathrm{t}(\mathrm{p} + t\omega, -\omega)\, \mathrm{d}t.$$

Its value can clearly be estimated by

$$\frac{1}{N} \sum_i^N \frac{\sigma_\mathrm{t}(\mathrm{p} + T_i\, \omega,\ -\omega)}{p(T_i)},$$

where random variables T_i are sampled from some distribution p. A natural approach is to apply stratified sampling, dividing the line from 0 to d into N strata and placing one random sample in each one. This could be done by transforming uniform random samples ξ by $d\xi$.

However, recall that Monte Carlo's strength compared to standard numerical integration algorithms comes when we are faced with high-dimensional integrals and discontinuous integrands. Here, we have a one-dimensional integral and an integrand that is often smoothly varying. (Consider the spatially varying density of a cloud or smoke, for example.) Pauly and collaborators have shown that a more efficient integration technique for this problem is to generate a stratified pattern where the sample inside each stratum has the same offset (Pauly 1999; Pauly, Kollig, and Keller 2000):

$$T_i = \frac{\xi + i}{N} d.$$

A single uniform random variable ξ is used for all of the samples. The result is that samples in adjacent strata can't be bunched together, as can happen with regular stratified sampling, thus leading to reduced variance.

The implementation of the `tau()` method here applies this technique, which is why it only takes a single random variable `u`. It also indirectly chooses the number of samples N to take for the estimator. The caller passes in `stepSize`, the desired size of the strata, and the method finds the length of the given ray overlapping the medium, d, and then determines N by

$$N = \left\lceil \frac{d}{\mathrm{stepSize}} \right\rceil.$$

N is not computed explicitly in the following implementation. Instead, the `while` loop will execute once for each sample taken until the point `t0` exits the region.

Mathematically, the overall estimate should be divided by N and the (constant) PDF value. However, the PDF is just $1/d = 1/(t_1 - t_0)$, so the division by Nd can be simply computed by multiplying by the step size at the end.

DensityRegion::tau() 733

⟨*Volume Scattering Definitions*⟩ +≡

```
Spectrum DensityRegion::tau(const Ray &r, float stepSize,
                            float u) const {
    float t0, t1;
    float length = r.d.Length();
    if (length == 0.f) return 0.f;
    Ray rn(r.o, r.d / length, r.mint * length, r.maxt * length, r.time);
    if (!IntersectP(rn, &t0, &t1)) return 0.;
    Spectrum tau(0.);
    t0 += u * stepSize;
    while (t0 < t1) {
        tau += sigma_t(rn(t0), -rn.d, r.time);
        t0 += stepSize;
    }
    return tau * stepSize;
}
```

FURTHER READING

Cook and collaborators first introduced random sampling for integration in rendering (Cook, Porter, and Carpenter 1984; Cook 1986), and Kajiya (1986) developed the general-purpose path-tracing algorithm. Other important early work on Monte Carlo in rendering includes Shirley's Ph.D. thesis (Shirley 1990) and articles by Arvo and Kirk on Russian roulette, splitting, and sources of bias in rendering algorithms (Kirk and Arvo 1991). The multiple importance sampling technique was invented by Veach and Guibas (Veach and Guibas 1995; Veach 1997).

Shirley and collaborators have written a number of key papers about Monte Carlo in graphics, including a survey of techniques for light source sampling (Shirley, Wang, and Zimmerman 1996) and recipes for warping uniform random numbers to the surfaces of various shapes (Shirley 1992). See also the "Further Reading" section in Chapter 15 for additional discussion of Monte Carlo techniques for direct lighting.

Mitchell (1996b) wrote a paper that investigates the effects of stratification for integration problems in graphics (including the two-dimensional problem of pixel antialiasing). In particular, he investigated the connection between the complexity of the function being integrated and the effect of stratification. In general, the smoother or simpler the function, the more stratification helps: for pixels with smooth variation over their areas or with just a few edges passing through them, stratification helps substantially, but as the complexity in a pixel is increased the gain from stratification is reduced. Nevertheless, because stratification never increases variance, there's no reason not to do it.

Keller and collaborators have written extensively on the application of quasi–Monte Carlo integration in graphics (Keller 1996, 2001; Friedel and Keller 2000; Kollig and Keller 2000, 2002), and Owen and Tribble (2005) discussed the requirements for correctly using quasi-random patterns with Metropolis sampling.

DensityRegion 591
Ray 66
Spectrum 263
Vector::Length() 62
VolumeRegion::
IntersectP() 588
VolumeRegion::sigma_t() 588

Lawrence et al. (2004) developed methods for sampling arbitrary BRDF models, including those based on measured reflectance data. They apply methods that factor the 4D BRDF into a product of two 2D functions, both of which are guaranteed to always be greater than zero, thus making it possible to use them as importance sampling distributions.

Subr and Arvo (2007) developed an efficient technique for sampling environment map light sources that accounts for both the $\cos\theta$ term from the scattering equation and also only generates samples in the hemisphere around the surface normal. A number of papers have been written about techniques transforming environment lights (like `InfiniteAreaLight`) into collections of distant light sources. See, for example, the papers by Agarwal et al. (2003) and Kollig and Keller (2003). This approach introduces additional correlation (using the same light sample positions at all shaded points), though the potential artifacts (banded shadows and reflections) may be less objectionable than Monte Carlo noise.

Talbot et al. (2005) applied *importance resampling* to rendering, showing that this variant of standard importance sampling is applicable to a number of problems in graphics, and Pegoraro et al. (2008a) implemented an approach that found sampling PDFs for global illumination over the course of rendering the image.

A number of researchers have recently developed efficient methods to directly sample the distribution given by the product of two functions (for example, the BSDF and an infinite area light source). See the "Further Reading" section in Chapter 15 for discussion of this work.

EXERCISES

❷ 14.1 One shortcoming of the current implementation of the `BSDF::Sample_f()` method is that if some of the `BxDF`s make a much larger contribution to the overall result than others, then uniformly choosing among them to determine a sampling distribution may be suboptimal. Modify this method so that it instead chooses among the `BxDF`s according to their relative reflectances. (Don't forget to also update `BSDF::Pdf()` to account for this change.) Can you create a contrived set of parameters to a `Material` that causes this approach to be substantially better than the built-in one? Does this change have a noticeable effect on Monte Carlo efficiency for typical scenes?

❸ 14.2 Implement the sampling method for arbitrary BRDFs introduced by Lawrence et al. (2005) as the sampling approach used for one or both of the measured BRDF implementations in Section 8.6. Perform tests using different sets of BRDF data and measure the improvement in Monte Carlo efficiency (variance divided by computation time) for different data sets.

❶ 14.3 Fix the buggy `Sphere::Sample()` and `Disk::Sample()` methods, which currently don't properly account for partial spheres and disks when they sample points on the surface. Create a scene that demonstrates the error from the current implementations and for which your solution is clearly an improvement.

BSDF::Pdf() 708
BSDF::Sample_f() 706
BxDF 428
Disk::Sample() 719
InfiniteAreaLight 629
Material 483
Sphere::Sample() 720

● 14.4 It is possible to derive a sampling method for cylinder area light sources that only chooses points over the visible area as seen from the receiving point, similar to the improved sphere sampling method in this chapter (Gardner et al. 1987; Zimmerman 1995). Learn more about these methods, or rederive them yourself, and write a new implementation of `Cylinder::Sample()` that implements such an algorithm. Verify that `pbrt` still generates correct images with your method and measure how much the improved version reduces variance for a fixed number of samples taken. How much does it improve efficiency? How do you explain any discrepancy between the amount of reduction in variance and the amount of improvement in efficiency?

● 14.5 Discuss situations where the current methods for sampling outgoing rays from `ProjectionLights` and `GonioPhotometricLights` may be extremely inefficient, choosing many rays in directions where the light source casts no illumination. Use the `Distribution2D` structure to implement improved sampling techniques for each of them based on sampling from a distribution based on the luminance in their two-dimensional image maps, properly accounting for the transformation from the 2D image map sampling distribution to the distribution of directions on the sphere. Verify that the system still computes the same images (modulo variance) with your new sampling techniques when using an `Integrator` that calls these methods. Determine how much efficiency is improved by using these sampling methods instead of the default ones.

● 14.6 Read the papers by Agarwal et al. (2003) or Kollig and Keller (2003) and implement one of these approaches to environment light sampling. Compare performance of the approach implemented in `pbrt` to this new approach. Discuss the trade-offs among them.

● 14.7 One useful technique not discussed in this chapter is the idea of adaptive density distribution functions that dynamically change the sampling distribution as samples are taken and information is available about the integrand's actual distribution as a result of evaluating the values of these samples. The standard Monte Carlo estimator can be written to work with a nonuniform distribution of random numbers used in a transformation method to generate samples X_i,

$$\sum_i^N \frac{f(X_i)}{p(X_i)p_r(\xi_i)},$$

just like the transformation from one sampling density to another. This leads to a useful sampling technique, where an algorithm can track which samples ξ_i were effective at finding large values of $f(x)$ and which weren't and then adjusts probabilities toward the effective ones (Booth 1986). A straightforward implementation would be to split $[0, 1]$ into bins of fixed width, track the average value of the integrand in each bin, and use this to change the distribution of ξ_i samples.

Investigate data structures and algorithms that support such sampling approaches and choose a sampling problem in `pbrt` to apply them to. Measure how well this approach works for the problem you selected. One difficulty with

Cylinder::Sample() 719
Distribution2D 672
GonioPhotometricLight 618
Integrator 740
ProjectionLight 614

methods like this is that different parts of the sampling domain will be the most effective at different times in different parts of the scene. For example, trying to adaptively change the sampling density of points over the surface of an area light source has to contend with the fact that, at different parts of the scene, different parts of the area light may be visible and thus be the important areas. You may find it useful to read Cline et al.'s paper (2008) on this topic.

2 14.8 `pbrt`'s routines for using complex shapes like triangle meshes or subdivision meshes as area light sources are written to handle simple shapes and do not scale well to shapes that refine into more than a few tens of triangles in the end. An effective way to sample shapes such as these is to uniformly sample rays from the cone that surrounds the shape as seen from the point being shaded, see if the ray hits the shape, and use the hit location, if any, as the sample point on the area source.

Implement such a sampling method for complex shapes. Test your implementation by using it to render images with simple shapes and then render images using complex shapes as area light sources.

2 14.9 If additional high-level information is provided about the characteristics of the scene, there are various ways that rendering can be done more efficiently. For example, consider an infinite area light representing the sky illuminating a room through windows. If the scene description also includes shapes that represent the window openings and if we know that all light paths from the sky to the interior of the room light go through a window, then another way to sample directions to the light source is to sample by area on the surfaces of the window shapes (represented as a collection of triangles, for example) and then use the corresponding directions to the light, converting from the measure over area to one over directions on the unit sphere, similarly to what is done for area lights attached to `Shapes`. This approach can save taking many samples and tracing the corresponding rays that would otherwise certainly be occluded by walls or other geometry.

Implement this approach as an optional `InfiniteAreaLight` sampling method in `pbrt` and measure the improvement in efficiency for rendering scenes where there is a substantial amount of occlusion between the part of the scene being rendered and the infinite light source. (Results can be improved by using this as yet another light sampling technique alongside the regular light source sampling and BSDF sampling used in Section 15.1.1 and then weighted using multiple importance sampling.)

2 14.10 The infinite area light importance sampling method implemented in this chapter doesn't perfectly match the distribution of the light source's emission distribution: recall that the emission function is computed with bilinear interpolation among image samples but the sampling distribution is computed as a piecewise-constant function of a slightly blurred version of the texture map. In some cases, this discrepancy can lead to high variance when there are localized extremely bright texels. (In the worst case, consider a source that has a very small value at all texels except one, which is 10,000 times brighter than all of the

`InfiniteAreaLight` 629
`Shape` 108

others.) In that case, the value of the function may be much higher than predicted by the sampling PDFs at points very close to that sample, thus leading to high variance ($f(x)/p(x)$ is large).

Construct an environment map where this problem manifests itself in `pbrt` and fix the system so that the excessive variance goes away. One option is to modify the system so that the sampling distribution and the illumination function match perfectly by point sampling the environment map for lookups, rather than using bilinear filtering, though this can lead to undesirable image artifacts in the environment map, especially when directly visible from the camera. Can you find a way to only use the point-sampled environment map for direct lighting calculations but to use bilinear filtering for camera rays and rays that have only undergone specular reflection?

The other alternative is to modify the sampling distribution so that it matches the bilinearly filtered environment map values perfectly. Exact 2D sampling distributions for bilinear functions can be computed, though finding the sample value corresponding to a random variable is more computationally expensive, requiring solving a quadratic equation. Can you construct a scene where this overhead is worthwhile in return for the resulting variance reduction?



CHAPTER FIFTEEN

15 LIGHT TRANSPORT I: SURFACE REFLECTION

This chapter brings together the ray-tracing algorithms, radiometric concepts, and Monte Carlo sampling algorithms of the previous chapters to implement a set of integrators that compute scattered radiance from surfaces in the scene. Integrators are so named because they are responsible for evaluating the integral equation that describes the equilibrium distribution of radiance in an environment (the light transport equation). As the `SamplerRenderer` uses the `Camera` to generate rays and then finds intersections with scene geometry, information about the intersections is passed to the `SurfaceIntegrator` and the `VolumeIntegrator` that the user selected; together these two classes are responsible for doing appropriate shading and lighting computations to compute the radiance along the ray, accounting for light reflected from the first surface visible along the ray as well as light attenuated and scattered by participating media along the ray. Surface integrators are covered in this chapter, and volume integrators are the topic of Chapter 16. This chapter concludes with the implementation of a `Renderer` that applies the Metropolis sampling approach introduced in Section 13.4 to the light transport problem.

Because the light transport equation can be solved in closed form only for trivial scenes, it's necessary to apply a numerical integration technique to approximate its solution. How best to do so has been an active area of research in rendering, and many solution methods have been proposed. In this chapter, we will present implementations of a number of different integrators based on Monte Carlo integration that represent a selection of representative approaches to the problem. Due to basic decisions made in `pbrt`'s design, this chapter does not include methods based on finite-element algorithms (e.g., radiosity), which is the other major approach to solving the light transport equation. See the "Further Reading" section for more information about this method.

The basic integrator interfaces are defined in core/integrator.h, and some utility functions used by integrators are in core/integrator.cpp. The implementations of the various integrators are in the integrators/ directory.

Both surface and volume integrators inherit from the Integrator abstract base class.

⟨*Integrator Declarations*⟩ ≡
```
class Integrator {
public:
    ⟨Integrator Interface 740⟩
};
```

The integrator implementation may optionally implement the Preprocess() method. It is called after the Scene has been fully initialized and gives the integrator a chance to do scene-dependent computation, such as allocating additional data structures that are dependent on the number of lights in the scene, or precomputing a rough representation of the distribution of radiance in the scene. Integrators that don't need to do anything along these lines can leave this method unimplemented.

⟨*Integrator Interface*⟩ ≡ 740
```
virtual void Preprocess(const Scene *scene, const Camera *camera,
                        const Renderer *renderer) {
}
```

If the integrator would like the Sampler to generate sample patterns in the Sample for it to use, it should implement the RequestSamples() method and call back to the Sample::Add1D() and Sample::Add2D() methods, as described in Section 7.2.1.

⟨*Integrator Interface*⟩ +≡ 740
```
virtual void RequestSamples(Sampler *sampler, Sample *sample,
                            const Scene *scene) {
}
```

The SurfaceIntegrator interface class adds a single required method beyond those in the Integrator.

⟨*Integrator Declarations*⟩ +≡
```
class SurfaceIntegrator : public Integrator {
public:
    ⟨SurfaceIntegrator Interface 741⟩
};
```

The key method that all surface integrators must implement is SurfaceIntegrator::Li(), which returns the outgoing radiance at the intersection point of a given ray with the scene geometry, or, equivalently, incident radiance at the origin of the ray. (Recall that radiance is unchanged as it passes through a vacuum; VolumeIntegrators will separately compute the effect of attenuation through participating media between the ray origin and the intersection point.)

The parameters to this method are the following:

- scene—A pointer to the Scene being rendered. The integrator will query the scene for information about the lights and geometry, and so on.

Camera 302
Integrator 740
Renderer 24
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sampler 340
Scene 22
SurfaceIntegrator 740
SurfaceIntegrator::Li() 741
VolumeIntegrator 876

- `renderer`—A pointer to the `Renderer` being used for rendering. The integrator may call the `Li()` or `Transmittance()` methods of the renderer, which in turn compute radiance and transmittance along arbitrary rays including both surface and volume scattering effects.[1]
- `ray`—The ray along which the incident radiance should be evaluated.
- `isect`—An `Intersection` object encapsulating information about the first intersection of the ray with the scene geometry.
- `sample`—A pointer to a `Sample` generated by the `Sampler` for this ray. Some integrators will use information in this structure for Monte Carlo sampling.
- `arena`—A `MemoryArena` for efficient temporary memory allocation by the integrator. The integrator should assume that any memory it allocates with the arena will be freed shortly after the `Li()` method returns and thus should not use the arena to allocate any memory that must persist for longer than is needed for the current ray.

The method returns a `Spectrum` that represents the radiance along the ray:

⟨*SurfaceIntegrator Interface*⟩ ≡ 740
```
virtual Spectrum Li(const Scene *scene, const Renderer *renderer,
    const RayDifferential &ray, const Intersection &isect,
    const Sample *sample, RNG &rng, MemoryArena &arena) const = 0;
```

15.1 DIRECT LIGHTING

Before we introduce the light transport equation in its full generality, we will implement an integrator that accounts for only direct lighting—light that has traveled directly from a light source to the point being shaded—and ignores indirect illumination from objects that are not themselves emissive. Starting out with this integrator allows us to focus on some of the key details of direct lighting without worrying about the full light transport equation. Furthermore, some of the routines developed here will be used again in subsequent integrators that solve the complete light transport equation. Figure 15.1 shows the San Miguel scene rendered with direct lighting only.

The implementation here provides two different strategies for computing direct lighting. Each method computes an unbiased estimate of exitant radiance at a point in a given direction. The `LightStrategy` enumerant records which approach has been selected. The first strategy loops over all of the lights and takes a number of samples based on `Light::nSamples` from each of them, summing the result. The second takes a single sample from just one of the lights, chosen at random.

Depending on the scene being rendered, either of these approaches may be more appropriate. For example, if many image samples are being taken for each pixel (e.g., due to depth of field), a single light sample may be more appropriate, since in the aggregate they will sample the direct lighting enough to give a high-quality image. Alternatively, if few image samples are being taken, sampling all lights may be preferable.

CreateRadianceProbes 958
Intersection 186
Light::nSamples 606
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Sampler 340
SamplerRenderer 25
Scene 22
Spectrum 263

1 The `Renderer` will usually be a `SamplerRenderer` instance, but other `Renderer` implementations like `CreateRadianceProbes` that use instances of `Integrators` from this chapter will be the `Renderer` in that case.



Figure 15.1: Scene Rendered with Direct Lighting Only. Because only direct lighting is considered, some portions of the image are completely black because they are only lit by indirect illumination. *(Model courtesy of Guillermo M. Leal Llaguno.)*

⟨*DirectLightingIntegrator Declarations*⟩ ≡
```
enum LightStrategy { SAMPLE_ALL_UNIFORM, SAMPLE_ONE_UNIFORM };
```

⟨*DirectLightingIntegrator Declarations*⟩ +≡
```
class DirectLightingIntegrator : public SurfaceIntegrator {
public:
    ⟨DirectLightingIntegrator Public Methods⟩
private:
    ⟨DirectLightingIntegrator Private Data 742⟩
};
```

The implementation of the constructor isn't included here. It initializes the lighting strategy with a value passed in.

⟨*DirectLightingIntegrator Private Data*⟩ ≡ 742
```
LightStrategy strategy;
```

The member variables that record the sample information for direct lighting are declared with a level of indirection in a separate fragment, so that other integrators later in this chapter that use the direct lighting functions defined here can easily incorporate the appropriate member variables.

⟨*DirectLightingIntegrator Private Data*⟩ +≡ 742 SurfaceIntegrator 740
```
    ⟨Declare sample parameters for light source sampling 743⟩
int lightNumOffset;
```

The number and types of samples needed by this integrator depend on the sampling strategy used: if a single sample is taken from a single light, then only one of each of the `LightSampleOffsets` and `BSDFSampleOffsets` object needs to be created; otherwise, one is allocated for each light source in the scene. These structures handle the mechanics of informing the sampler of how many samples and of what dimensionality are needed—they are defined in Sections 14.6.1 and 14.5.6, respectively. Sample patterns are needed both to select points on light sources and to select BSDF directions for the direct lighting computation; both sampling approaches are combined with multiple importance sampling in the implementation below.

⟨*Declare sample parameters for light source sampling*⟩ ≡ 742, 788, 803
```
LightSampleOffsets *lightSampleOffsets;
BSDFSampleOffsets *bsdfSampleOffsets;
```

⟨*DirectLightingIntegrator Method Definitions*⟩ ≡
```
void DirectLightingIntegrator::RequestSamples(Sampler *sampler,
        Sample *sample, const Scene *scene) {
    if (strategy == SAMPLE_ALL_UNIFORM) {
        ⟨Allocate and request samples for sampling all lights 743⟩
        lightNumOffset = -1;
    }
    else {
        ⟨Allocate and request samples for sampling one light 744⟩
    }
}
```

If all of the lights are being sampled, the integrator needs `nLights` individual sampling patterns from the `Sampler`, each one sized according to the number of samples in that light's `nSamples` member variable. Note that the integrator only uses that value as a starting point. The `Sampler::RoundSize()` method is given an opportunity to change that value to a more appropriate one based on its particular sample generation technique. (For example, some samplers only generate collections of samples with power-of-two sizes.)

⟨*Allocate and request samples for sampling all lights*⟩ ≡ 743, 774, 788, 803
```
uint32_t nLights = scene->lights.size();
lightSampleOffsets = new LightSampleOffsets[nLights];
bsdfSampleOffsets = new BSDFSampleOffsets[nLights];
for (uint32_t i = 0; i < nLights; ++i) {
    const Light *light = scene->lights[i];
    int nSamples = light->nSamples;
    if (sampler) nSamples = sampler->RoundSize(nSamples);
    lightSampleOffsets[i] = LightSampleOffsets(nSamples, sample);
    bsdfSampleOffsets[i] = BSDFSampleOffsets(nSamples, sample);
}
```

Things are easier if only a single sample needs to be taken from a single light. Other than the immediate differences from this change, the only other difference is that an additional 1D sample is needed to choose which light to sample.

BSDFSampleOffsets 706
DirectLightingIntegrator::bsdfSampleOffsets 743
DirectLightingIntegrator::lightNumOffset 742
DirectLightingIntegrator::lightSampleOffsets 743
DirectLightingIntegrator::SAMPLE_ALL_UNIFORM 742
DirectLightingIntegrator::strategy 742
Light 606
LightSampleOffsets 710
Sample 343
Sampler 340
Sampler::RoundSize() 344
Scene 22
Scene::lights 23

⟨*Allocate and request samples for sampling one light*⟩ ≡ 743
```
lightSampleOffsets = new LightSampleOffsets[1];
lightSampleOffsets[0] = LightSampleOffsets(1, sample);
lightNumOffset = sample->Add1D(1);
bsdfSampleOffsets = new BSDFSampleOffsets[1];
bsdfSampleOffsets[0] = BSDFSampleOffsets(1, sample);
```

The general form of the `DirectLightingIntegrator::Li()` method is similar to that of `WhittedIntegrator::Li()`. The BSDF at the intersection point is computed, emitted radiance is added if the surface is emissive, and so on. We won't include the full implementation of `DirectLightingIntegrator::Li()` here in order to focus on its key fragment, ⟨*Compute direct lighting at hit point*⟩, which estimates the value of the integral that gives the reflected radiance.

The scattering equation from Section 5.6 says that exitant radiance $L_o(p, \omega_o)$ from a point p on a surface in direction ω_o is the sum of emitted radiance from the surface at the point plus the integral of incoming radiance over the sphere times the BSDF for each direction and a cosine term. For the `DirectLightingIntegrator`, we are only interested in incident radiance directly from light sources, which we denote by $L_d(p, \omega)$:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i)\, L_d(p, \omega_i)\, |\cos \theta_i|\, d\omega_i. \quad [15.1]$$

The value of $L_e(p, \omega_o)$ is found by calling the `Intersection::Le()` method at the intersection point. To estimate the integral over the sphere, we will apply Monte Carlo integration.

⟨*Compute direct lighting for* `DirectLightingIntegrator` *integrator*⟩ ≡
```
if (scene->lights.size() > 0) {
    ⟨Apply direct lighting strategy⟩
}
```

The fragment ⟨*Apply direct lighting strategy*⟩ is also not included here. It just calls one of the functions that implement the direct lighting approaches—`UniformSampleAllLights()` or `UniformSampleOneLight()`—depending on the value of the `strategy` member variable.

To understand the approaches implemented by the different strategies, first consider the part of the direct lighting equation that we're concerned with here:

$$\int_{S^2} f(p, \omega_o, \omega_i)\, L_d(p, \omega_i)\, |\cos \theta_i|\, d\omega_i.$$

This can be broken into a sum over the lights in the scene

$$\sum_{j=1}^{\text{lights}} \int_{S^2} f(p, \omega_o, \omega_i)\, L_{d(j)}(p, \omega_i)\, |\cos \theta_i|\, d\omega_i,$$

where $L_{d(j)}$ denotes incident radiance from the jth light and

$$L_d(p, \omega_i) = \sum_j L_{d(j)}(p, \omega_i).$$

BSDFSampleOffsets 706
DirectLightingIntegrator::
bsdfSampleOffsets 743
DirectLightingIntegrator::
Li() 744
DirectLightingIntegrator::
lightNumOffset 742
DirectLightingIntegrator::
lightSampleOffsets 743
Intersection::Le() 625
LightSampleOffsets 710
Sample::Add1D() 344
Scene::lights 23
UniformSampleAllLights() 745
UniformSampleOneLight() 746
WhittedIntegrator::Li() 42

One valid approach is to estimate each term of this sum individually, adding the results together. This is the most basic direct lighting strategy and is implemented in `UniformSampleAllLights()`, which we have implemented as a global function rather than a `DirectLightingIntegrator` method so that other integrators can use it as well.

Note that the number of samples for each light is *not* taken from its `nSamples` member variable but instead from the number of samples available in the `Sample` (the value of which, in turn, was returned by the earlier `Sampler::RoundSize()` call). Note also that the `lightSampleOffsets` and `bsdfSampleOffsets` parameters may be `NULL`, in which case uniform random samples are used for the lighting calculation instead.

⟨*Integrator Utility Functions*⟩ +≡
```
Spectrum UniformSampleAllLights(const Scene *scene,
        const Renderer *renderer, MemoryArena &arena, const Point &p,
        const Normal &n, const Vector &wo, float rayEpsilon,
        float time, BSDF *bsdf, const Sample *sample, RNG &rng,
        const LightSampleOffsets *lightSampleOffsets,
        const BSDFSampleOffsets *bsdfSampleOffsets) {
    Spectrum L(0.);
    for (uint32_t i = 0; i < scene->lights.size(); ++i) {
        Light *light = scene->lights[i];
        int nSamples = lightSampleOffsets ?
                       lightSampleOffsets[i].nSamples : 1;
        ⟨Estimate direct lighting from light samples 745⟩
    }
    return L;
}
```

BSDF 478
BSDFSample 705
BSDFSampleOffsets 706
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
EstimateDirect() 749
Light 606
LightSample 710
LightSampleOffsets 710
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Sample 343
Sampler::RoundSize() 344
Scene 22
Scene::lights 23
Spectrum 263
UniformSampleAllLights() 745
Vector 57

For each light sample the `EstimateDirect()` function, to be defined shortly, computes the value of the Monte Carlo estimator for its contribution. All that has to be done here is to average the values of the estimates.

⟨*Estimate direct lighting from* `light` *samples*⟩ ≡ 745
```
Spectrum Ld(0.);
for (int j = 0; j < nSamples; ++j) {
    ⟨Find light and BSDF sample values for direct lighting estimate 746⟩
    Ld += EstimateDirect(scene, renderer, arena, light, p, n, wo,
        rayEpsilon, time, bsdf, rng, lightSample, bsdfSample,
        BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
}
L += Ld / nSamples;
```

Before calling `EstimateDirect()`, this function assembles the values for the various random samples that will be used for Monte Carlo integration. The `LightSample` and `BSDFSample` structures encapsulate the collections of random values that the lights and BSDFs need for their sampling routines; their constructors take a `Sample`, the respective `SampleOffsets` types, and the sample number and extract the appropriate sample values

from the Sample structure. If the SampleOffsets types are NULL, then a reference to a random number generator is passed to the sample constructors so that they can generate uniform random samples.

⟨*Find light and BSDF sample values for direct lighting estimate*⟩ ≡ 745
```
LightSample lightSample;
BSDFSample bsdfSample;
if (lightSampleOffsets != NULL && bsdfSampleOffsets != NULL) {
    lightSample = LightSample(sample, lightSampleOffsets[i], j);
    bsdfSample = BSDFSample(sample, bsdfSampleOffsets[i], j);
}
else {
    lightSample = LightSample(rng);
    bsdfSample = BSDFSample(rng);
}
```

In a scene with a large number of lights, it may not be desirable to always compute direct lighting from all of the lights at every point that is shaded. Monte Carlo gives a way to do this that still computes the correct result on average. Consider as an example computing the expected value of the sum of two functions $E[f(x) + g(x)]$. If we randomly evaluate just one of $f(x)$ or $g(x)$ and multiply the result by two, then the expected value of the result will still be $f(x) + g(x)$. In fact, this generalizes to sums of N terms. This is a straightforward application of conditional probability; see Ross (2002, p. 102) for a proof. Here we estimate direct lighting for only one randomly chosen light and multiply the result by the number of lights to compensate.

⟨*Integrator Utility Functions*⟩ +≡
```
Spectrum UniformSampleOneLight(const Scene *scene,
        const Renderer *renderer, MemoryArena &arena, const Point &p,
        const Normal &n, const Vector &wo, float rayEpsilon, float time,
        BSDF *bsdf, const Sample *sample, RNG &rng, int lightNumOffset,
        const LightSampleOffsets *lightSampleOffset,
        const BSDFSampleOffsets *bsdfSampleOffset) {
    ⟨Randomly choose a single light to sample, light 747⟩
    ⟨Initialize light and bsdf samples for single light sample 747⟩
    return (float)nLights *
        EstimateDirect(scene, renderer, arena, light, p, n, wo,
                       rayEpsilon, time, bsdf, rng, lightSample,
                       bsdfSample, BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
}
```

Which of the nLights to sample illumination from is determined using the random sample available in the 1D integrator sample lightNumOffset, if available. Although this is never the case for the DirectLightingIntegrator integrator, sometimes other integrators will call this function without such a sample available. In that case, they pass −1 for the sample offset and a uniform random number is used instead.

BSDF 478
BSDFSample 705
BSDFSampleOffsets 706
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DirectLightingIntegrator 742
EstimateDirect() 749
LightSample 710
LightSampleOffsets 710
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Sample 343
Scene 22
Spectrum 263
Vector 57

⟨*Randomly choose a single light to sample,* `light`⟩ ≡ 746
```
int nLights = int(scene->lights.size());
if (nLights == 0) return Spectrum(0.);
int lightNum;
if (lightNumOffset != -1)
    lightNum = Floor2Int(sample->oneD[lightNumOffset][0] * nLights);
else
    lightNum = Floor2Int(rng.RandomFloat() * nLights);
lightNum = min(lightNum, nLights-1);
Light *light = scene->lights[lightNum];
```

It's possible to be even more creative in choosing the individual light sampling probabilities than the uniform method used in `UniformSampleOneLight()`. In fact, we're free to set the probabilities any way we like, so long as we weight the result appropriately and there is a nonzero probability of sampling any light that contributes to the reflection at the point. The better a job we do at setting the probabilities, the more efficient the Monte Carlo estimator will be, and the fewer rays will be needed to lower variance to an acceptable level. (This is just the discrete instance of importance sampling.) One widely used approach is to base the sample distribution on the total power of each light. In a similar manner, we could take more than one light sample with such an approach; indeed, any number of samples can be taken in the end, so long as they are weighted appropriately. This topic is discussed further in Exercise 15.5 at the end of the chapter.

⟨*Initialize light and bsdf samples for single light sample*⟩ ≡ 746
```
LightSample lightSample;
BSDFSample bsdfSample;
if (lightSampleOffset != NULL && bsdfSampleOffset != NULL) {
    lightSample = LightSample(sample, *lightSampleOffset, 0);
    bsdfSample = BSDFSample(sample, *bsdfSampleOffset, 0);
}
else {
    lightSample = LightSample(rng);
    bsdfSample = BSDFSample(rng);
}
```

15.1.1 ESTIMATING THE DIRECT LIGHTING INTEGRAL

Having chosen a particular light to estimate direct lighting from, we need to estimate the value of the integral

$$\int_{S^2} f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{d}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}$$

for that light. To compute this estimate, we need to choose one or more directions ω_j and apply the Monte Carlo estimator:

$$\frac{1}{N}\sum_{j=1}^{N} \frac{f(\mathrm{p}, \omega_\mathrm{o}, \omega_j)\, L_\mathrm{d}(\mathrm{p}, \omega_j)\, |\cos\theta_j|}{p(\omega_j)}.$$

BSDFSample 705
Floor2Int() 1002
Light 606
LightSample 710
RNG::RandomFloat() 1003
Sample::oneD 344
Scene::lights 23
Spectrum 263
UniformSampleOneLight() 746

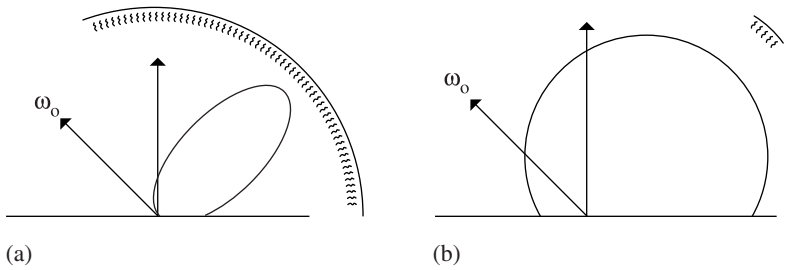


Figure 15.2: Depending on the actual distributions of the BSDF and the light source as a function of direction from the point being illuminated, one or the other may be a much more effective distribution to draw samples from for importance sampling. (a) Sampling the BSDF will generally be more effective, since illumination is coming from many directions but the BSDF's value is large for only a few of them. (b) The converse is true: sampling the BSDF in this case would select many directions along which no incident illumination was arriving.

To reduce variance, we will use importance sampling to choose the directions ω_j. Because both the BSDF and the direct radiance terms are individually complex, it can be difficult to find sampling distributions that match their product well. (However, see the "Further Reading" section as well as Exercise 15.3 at the end of this chapter for approaches for sampling their product directly.) Here we will use the BSDF's sampling distribution for some of the samples and the light's for the rest. Depending on the characteristics of each of them, one of these two sampling methods may be far more effective than the other. Therefore, we will use multiple importance sampling to reduce variance for the cases where one or the other is more effective.

Figure 15.2 shows two cases where one of the sampling methods is much better than the other. In Figure 15.2(a), the BSDF is very specular, and the light source is relatively large. Sampling the BSDF will be effective at finding directions where the integrand's value is large, while sampling the light will be less effective: most of the samples will not contribute much since the BSDF is small for most of the directions to the light source. When the light happens to sample a point in the BSDF's glossy region, there is a spike in the image because the light returns a low PDF, while the value of the integrand is relatively large. As a result, the variance is high because the sampling distribution didn't match the actual distribution of the function's values very well.

On the other hand, sometimes sampling the light is the right strategy. In Figure 15.2(b), the BSDF is less highly specular and thus nonzero over many directions, while the light is relatively small. It will be far more effective to choose points on the light to select the incident direction since the BSDF will have trouble finding directions where there is nonzero incident radiance from the light. Similarly to the first case, we would encounter substantial variance since the sampling distribution didn't match the overall function well.

By applying multiple importance sampling, not only can we use both of the two sampling methods, but we can also do so in a way that eliminates the extreme variance from these two situations, since the weighting terms from MIS reduce this variance substantially.

⟨*Integrator Utility Functions*⟩ +≡
```
Spectrum EstimateDirect(const Scene *scene, const Renderer *renderer,
        MemoryArena &arena, const Light *light, const Point &p,
        const Normal &n, const Vector &wo, float rayEpsilon, float time,
        const BSDF *bsdf, RNG &rng, const LightSample &lightSample,
        const BSDFSample &bsdfSample, BxDFType flags) {
    Spectrum Ld(0.);
    ⟨Sample light source with multiple importance sampling 749⟩
    ⟨Sample BSDF with multiple importance sampling 750⟩
    return Ld;
}
```

First, one sample is taken with the light's sampling distribution using `Sample_L()`, which also returns the light's emitted radiance and the value of the PDF for the sampled direction. Only if the light successfully samples a direction and returns nonzero emitted radiance does the function here go ahead and evaluate the BSDF for the two directions; otherwise, there's no reason to go through the computational expense. For example, a spotlight returns zero radiance for points outside its illumination cone. Then only if the BSDF's value is nonzero is a shadow ray traced to check for occlusion.

⟨*Sample light source with multiple importance sampling*⟩ ≡ 749
```
Vector wi;
float lightPdf, bsdfPdf;
VisibilityTester visibility;
Spectrum Li = light->Sample_L(p, rayEpsilon, lightSample, time,
                              &wi, &lightPdf, &visibility);
if (lightPdf > 0. && !Li.IsBlack()) {
    Spectrum f = bsdf->f(wo, wi, flags);
    if (!f.IsBlack() && visibility.Unoccluded(scene)) {
        ⟨Add light's contribution to reflected radiance 750⟩
    }
}
```

BSDF 478
BSDF::f() 481
BSDF::Pdf() 708
BSDFSample 705
BxDFType 428
Light 606
Light::Sample_L() 608
LightSample 710
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Scene 22
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VisibilityTester 608
VisibilityTester::
 Unoccluded() 609

Once it is known that the light is visible and radiance is arriving at the point, the value of the Monte Carlo estimator can be computed. First, radiance from the light to the illuminated point is scaled by the beam transmittance between the two points to account for attenuation due to participating media. Next, recall from Section 14.6.2 that if the light is described by a delta distribution then there is an implied delta distribution in both the emitted radiance value returned from `Sample_L()` as well as the PDF and that they are expected to cancel out when the estimator is evaluated. In this case, we must not try to apply multiple importance sampling and should compute the standard estimator instead. If this isn't a delta distribution light source, then the `BSDF::Pdf()` method is called to return the BSDF's PDF value for sampling the direction ω_i, and the MIS estimator is used, where the weight is computed here with the power heuristic.

⟨*Add light's contribution to reflected radiance*⟩ ≡ 749
```
Li *= visibility.Transmittance(scene, renderer, NULL, rng, arena);
if (light->IsDeltaLight())
    Ld += f * Li * (AbsDot(wi, n) / lightPdf);
else {
    bsdfPdf = bsdf->Pdf(wo, wi, flags);
    float weight = PowerHeuristic(1, lightPdf, 1, bsdfPdf);
    Ld += f * Li * (AbsDot(wi, n) * weight / lightPdf);
}
```

In a similar manner, a sample is now generated using the BSDF's sampling distribution. This step can be skipped if the light source is a delta distribution because, in that case, there's no chance that sampling the BSDF will give a direction that receives light from the source. If this is not the case, the BSDF can be sampled. One important detail is that the light's PDF and the multiple importance sampling weight are only computed if the BSDF component used for sampling ω_i is non specular; in the specular case, MIS shouldn't be applied since there is no chance of the light sampling the specular direction.

⟨*Sample BSDF with multiple importance sampling*⟩ ≡ 749
```
if (!light->IsDeltaLight()) {
    BxDFType sampledType;
    Spectrum f = bsdf->Sample_f(wo, &wi, bsdfSample, &bsdfPdf, flags,
                                &sampledType);
    if (!f.IsBlack() && bsdfPdf > 0.) {
        float weight = 1.f;
        if (!(sampledType & BSDF_SPECULAR)) {
            lightPdf = light->Pdf(p, wi);
            if (lightPdf == 0.)
                return Ld;
            weight = PowerHeuristic(1, bsdfPdf, 1, lightPdf);
        }
        ⟨Add light contribution from BSDF sampling 750⟩
    }
}
```

Given a direction sampled by the BSDF, we need to find out if the ray along that direction intersects this particular light source, and if so, how much radiance from the light reaches the surface. The code must account for both regular area lights, with geometry associated with them, as well as lights like the `InfiniteAreaLight` that don't have geometry but need to return their radiance for the sample ray via the `Light::Le()` method.

⟨*Add light contribution from BSDF sampling*⟩ ≡ 750
```
Intersection lightIsect;
Spectrum Li(0.f);
RayDifferential ray(p, wi, rayEpsilon, INFINITY, time);
if (scene->Intersect(ray, &lightIsect)) {
    if (lightIsect.primitive->GetAreaLight() == light)
        Li = lightIsect.Le(-wi);
}
```

AbsDot() 61
BSDF::Pdf() 708
BSDF::Sample_f() 706
BSDF_SPECULAR 428
BxDFType 428
GeometricPrimitive::GetAreaLight() 188
InfiniteAreaLight 629
INFINITY 1002
Intersection 186
Intersection::Le() 625
Intersection::primitive 186
Light::IsDeltaLight() 608
Light::Le() 631
Light::Pdf() 711
PowerHeuristic() 693
RayDifferential 69
Renderer::Transmittance() 25
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
VisibilityTester::Transmittance() 609

```
else
    Li = light->Le(ray);
if (!Li.IsBlack()) {
    Li *= renderer->Transmittance(scene, ray, NULL, rng, arena);
    Ld += f * Li * AbsDot(wi, n) * weight / bsdfPdf;
}
```

15.2 THE LIGHT TRANSPORT EQUATION

The light transport equation (LTE) is the governing equation that describes the equilibrium distribution of radiance in a scene. It gives the total reflected radiance at a point on a surface in terms of emission from the surface, its BSDF, and the distribution of incident illumination arriving at the point. The key task of the `Integrator` objects in `pbrt` is to numerically compute a solution to the LTE to find the incident radiance arriving at the camera. For now we will consider the case where there are no participating media in the scene. Chapter 16 describes the generalizations to this process necessary for scenes that do have participating media.

The detail that makes evaluating the LTE difficult is the fact that incident radiance at a point is affected by the geometry and scattering properties of all of the objects in the scene. For example, a bright light shining on a red object may cause a reddish tint on nearby objects in the scene, or glass may focus light into caustic patterns on a tabletop. Rendering algorithms that account for this complexity are often called *global illumination* algorithms, to differentiate them from *local illumination* algorithms that use only information about the local surface properties in their shading computations.

In this section, we will first derive the LTE and describe some approaches for manipulating the equation to make it easier to solve numerically. We will then describe two generalizations of the LTE that make some of its key properties more clear and serve as the foundation for some of the advanced integrators we will implement later in this chapter.

15.2.1 BASIC DERIVATION

The light transport equation depends on the basic assumptions we have already made in choosing to use radiometry to describe light—that wave optics effects are unimportant and that the distribution of radiance in the scene is in equilibrium. To compute radiance arriving at the film along a camera ray, we would like to express the exitant radiance from the camera ray's intersection point p in the direction ω_o from p to the film sample. We will denote this radiance measurement by $L_o(p, \omega_o)$.

The key principle underlying the LTE is *energy balance*. Any change in energy has to be "charged" to some process, and we must keep track of all the energy. Since we are assuming that lighting is a linear process, the difference between the amount of energy coming in and energy going out of a system must also be equal to the difference between energy emitted and energy absorbed. This idea holds at many levels of scale. On a macro level we have conservation of power:

Integrator 740

$$\Phi_o - \Phi_i = \Phi_e - \Phi_a.$$

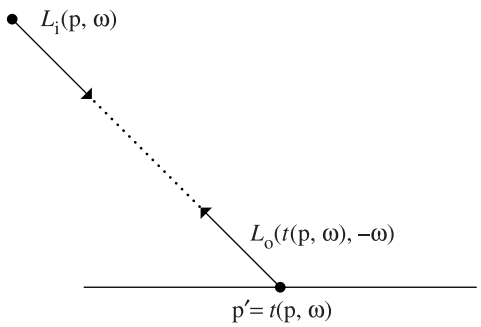


Figure 15.3: Radiance along a Ray through Free Space Is Unchanged. Therefore, to compute the incident radiance along a ray from point p in direction ω, we can find the first surface the ray intersects and compute exitant radiance in the direction $-\omega$ there. The trace operator $t(p, \omega)$ gives the point p' on the first surface that the ray (p, ω) intersects.

The difference between the power leaving an object, Φ_o, and the power entering it, Φ_i, is equal to the difference between the power it emits and the power it absorbs, $\Phi_e - \Phi_a$.

In order to enforce energy balance at a surface, exitant radiance must be equal to emitted radiance plus the fraction of incident radiance that is scattered. Emitted radiance is given by L_e, and scattered radiance is given by the scattering equation, which gives

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos \theta_i|\, d\omega_i.$$

Because we have assumed for now that no participating media are present, radiance is constant along rays through the scene. We can therefore relate the incident radiance at p to the outgoing radiance from another point p', as shown by Figure 15.3. If we define the *ray-casting function* $t(p, \omega)$ as a function that computes the first surface point p' intersected by a ray from p in the direction ω, we can write the incident radiance at p in terms of outgoing radiance at p':

$$L_i(p, \omega) = L_o(t(p, \omega), -\omega).$$

In case the scene is not closed, we will define the ray-casting function to return a special value Λ if the ray (p, ω) doesn't intersect any object in the scene, such that $L_o(\Lambda, \omega)$ is always zero.

Dropping the subscripts from L_o for brevity, this relationship allows us to write the LTE as

$$L(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i)\, L(t(p, \omega_i), -\omega_i)\, |\cos \theta_i|\, d\omega_i. \quad [15.2]$$

The key to the above representation is that there is only *one* quantity of interest, exitant radiance from points on surfaces. Of course, it appears on both sides of the equation, so our task is still not simple, but it is certainly better. It is important to keep in mind that we were able to arrive at this equation simply by enforcing energy balance in our scene.

15.2.2 ANALYTIC SOLUTIONS TO THE LTE

The brevity of the LTE belies the fact that it is impossible to solve analytically in general. The complexity that comes from physically based BSDF models, arbitrary scene geometry, and the intricate visibility relationships between objects all conspire to mandate a numerical solution technique. Fortunately, the combination of ray-tracing algorithms and Monte Carlo integration gives a powerful pair of tools that can handle this complexity without needing to impose restrictions on various components of the LTE (e.g., requiring that all BSDFs be Lambertian or substantially limiting the geometric representations that are supported).

It is possible to find analytic solutions to the LTE in extremely simple settings. While this is of little help for general-purpose rendering, it can help with debugging the implementations of integrators. If an integrator that is supposed to solve the complete LTE doesn't compute a solution that matches an analytic solution, then clearly there is a bug in the integrator. As an example, consider the interior of a sphere where all points on the surface of the sphere have a Lambertian BRDF, $f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i}) = c$, and also emit a constant amount of radiance in all directions. We have

$$L(\mathrm{p}, \omega_\mathrm{o}) = L_\mathrm{e} + c \int_{\mathcal{H}^2(\mathbf{n})} L(t(\mathrm{p}, \omega_\mathrm{i}), -\omega_\mathrm{i}) \, |\cos \theta_\mathrm{i}| \, \mathrm{d}\omega_\mathrm{i}.$$

The outgoing radiance distribution at any point on the sphere interior must be the same as at any other point; nothing in the environment could introduce any variation among different points. Therefore, the incident radiance distribution must be the same at all points, and the cosine-weighted integral of incident radiance must be the same everywhere as well. As such, we can replace the radiance functions with constants and simplify, writing the LTE as

$$L = L_\mathrm{e} + c\pi L.$$

While we could immediately solve this equation for L, it's interesting to consider successive substitution of the right-hand side into the L term on the right-hand side. If we also replace πc with ρ_hh, the reflectance of a Lambertian surface, we have

$$\begin{aligned} L &= L_\mathrm{e} + \rho_\mathrm{hh}(L_\mathrm{e} + \rho_\mathrm{hh}(L_\mathrm{e} + \cdots \\ &= \sum_{i=0}^{\infty} L_\mathrm{e} \rho_\mathrm{hh}^i. \end{aligned}$$

In other words, exitant radiance is equal to the emitted radiance at the point plus light that has been scattered by a BSDF once after emission, plus light that has been scattered twice, and so forth.

Because $\rho_\mathrm{hh} < 1$ due to conservation of energy, the series converges and the reflected radiance at all points in all directions is

$$L = \frac{L_\mathrm{e}}{1 - \rho_\mathrm{hh}}.$$

This process of repeatedly substituting the LTE's right-hand side into the incident radiance term in the integral can be instructive in more general cases.[2] For example, the `DirectLightingIntegrator` integrator effectively computes the result of making a single substitution:

$$L(\mathrm{p}, \omega_\mathrm{o}) = L_\mathrm{e}(\mathrm{p}, \omega_\mathrm{o}) + \int_{\mathcal{S}^2} f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{d}\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i},$$

where

$$L_\mathrm{d} = L_\mathrm{e}(t(\mathrm{p}, \omega_\mathrm{i}), -\omega_\mathrm{i}) + \int_{\mathcal{S}^2} f(t(\mathrm{p}, \omega_\mathrm{i}), \omega') L(t(t(\mathrm{p}, \omega_\mathrm{i}), \omega'), -\omega') |\cos\theta'|\, \mathrm{d}\omega'$$

and then ignores the result of multiply scattered light.

Over the next few pages, we will see how performing successive substitutions in this manner and then regrouping the results expresses the LTE in a more natural way for developing rendering algorithms.

15.2.3 THE SURFACE FORM OF THE LTE

One reason why the LTE as written in Equation (15.2) is complex is that the relationship between geometric objects in the scene is implicit in the ray-tracing operator $t(\mathrm{p}, \omega)$. Making the behavior of the ray-tracing operator explicit in the integrand will shed some light on the structure of this equation. To do this, we will rewrite Equation (15.2) as an integral over *area* instead of an integral over directions on the sphere.

First, we define exitant radiance from a point p' to a point p by

$$L(\mathrm{p}' \to \mathrm{p}) = L(\mathrm{p}', \omega)$$

if p' and p are mutually visible and $\omega = \mathrm{p} \widehat{-} \mathrm{p}'$. We can also write the BSDF at p' as

$$f(\mathrm{p}'' \to \mathrm{p}' \to \mathrm{p}) = f(\mathrm{p}', \omega_\mathrm{o}, \omega_\mathrm{i})$$

where $\omega_\mathrm{i} = \mathrm{p}'' \widehat{-} \mathrm{p}'$ and $\omega_\mathrm{o} = \mathrm{p} \widehat{-} \mathrm{p}'$ (Figure 15.4).

Rewriting the terms in the LTE in this manner isn't quite enough, however. We also need to multiply by the Jacobian that relates solid angle to area in order to transform the LTE from an integral over direction to one over surface area. Recall that this is $|\cos\theta'|/r^2$.

We will combine this change-of-variables term, the original $|\cos\theta|$ term from the LTE, and also a binary visibility function V ($V = 1$ if the two points are mutually visible, and $V = 0$ otherwise) into a single geometric coupling term, $G(\mathrm{p} \leftrightarrow \mathrm{p}')$:

$$G(\mathrm{p} \leftrightarrow \mathrm{p}') = V(\mathrm{p} \leftrightarrow \mathrm{p}') \frac{|\cos\theta|\, |\cos\theta'|}{\| \mathrm{p} - \mathrm{p}' \|^2}. \qquad [15.3]$$

2 Indeed, this sort of series expansion and inversion can be used in the general case, where quantities like the BSDF are expressed in terms of general operators that map incident radiance functions to exitant radiance functions. This approach forms the foundation for applying sophisticated tools from analysis to the light transport problem. See Arvo's thesis (Arvo 1995) and Veach's thesis (Veach 1997) for further information.

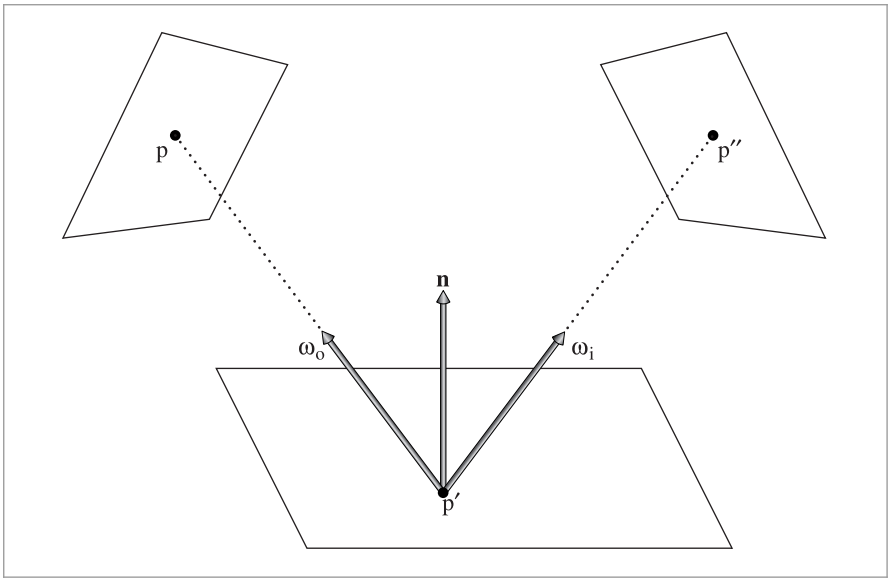


Figure 15.4: The three-point form of the light transport equation converts the integral to be over the domain of points on surfaces in the scene, rather than over directions over the sphere. It is a key transformation for deriving the path integral form of the light transport equation.

Substituting these into the light transport equation and converting to an area integral, we have

$$L(\mathrm{p}' \to \mathrm{p}) = L_\mathrm{e}(\mathrm{p}' \to \mathrm{p}) + \int_A f(\mathrm{p}'' \to \mathrm{p}' \to \mathrm{p}) L(\mathrm{p}'' \to \mathrm{p}') G(\mathrm{p}'' \leftrightarrow \mathrm{p}') \, \mathrm{d}A(\mathrm{p}''), \tag{15.4}$$

where A is all of the surfaces of the scene.

Although Equations (15.2) and (15.4) are equivalent, they represent two different ways of approaching light transport. To evaluate Equation (15.2) with Monte Carlo, we would sample a number of directions from a distribution of directions on the sphere and cast rays to evaluate the integrand. For Equation (15.4), however, we would choose a number of *points* on surfaces according to a distribution over surface area and compute the coupling between those points to evaluate the integrand, tracing rays to evaluate the visibility term $V(\mathrm{p} \leftrightarrow \mathrm{p}')$.

15.2.4 INTEGRAL OVER PATHS

To go from the area integral to a more flexible form of the LTE, a sum over light-carrying paths of different lengths, we can now start to expand the three-point light transport equation, repeatedly substituting the right-hand side of the equation into the $L(\mathrm{p}' \to \mathrm{p}')$ term inside the integral. Here are the first few terms that give incident radiance at a point p_0 from another point p_1, where p_1 is the first point on a surface along the ray from p_0 in

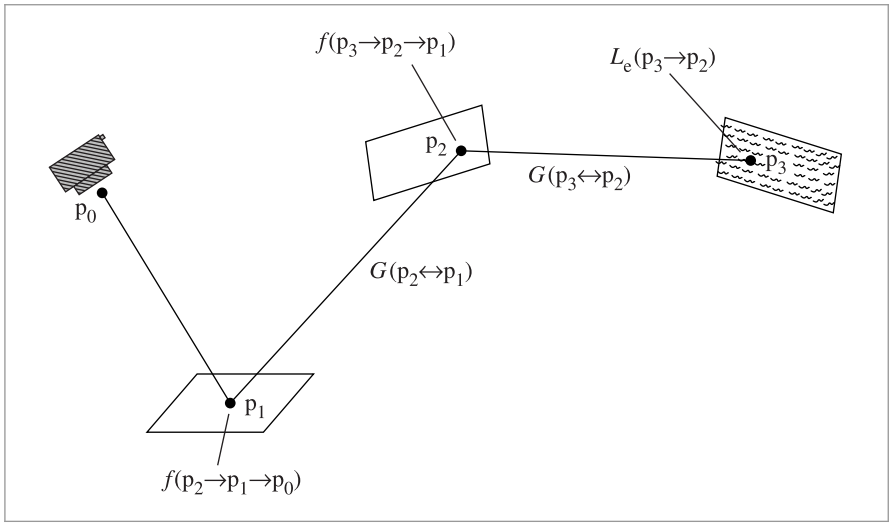


Figure 15.5: The integral over all points p_2 and p_3 on surfaces in the scene given by the light transport equation gives the total contribution of two bounce paths to radiance leaving p_1 in the direction of p_0. The components of the product in the integrand are shown here: the emitted radiance from the light, L_e; the geometric terms between vertices, G; and scattering from the BSDFs, f.

direction $p_1 - p_0$:

$$\begin{aligned} L(p_1 \to p_0) = {} & L_e(p_1 \to p_0) \\ & + \int_A L_e(p_2 \to p_1) f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) \mathrm{d}A(p_2) \\ & + \int_A \int_A L_e(p_3 \to p_2) f(p_3 \to p_2 \to p_1) G(p_3 \leftrightarrow p_2) \\ & \qquad \times f(p_2 \to p_1 \to p_0) G(p_2 \leftrightarrow p_1) \, \mathrm{d}A(p_3) \, \mathrm{d}A(p_2) + \cdots \end{aligned}$$

Each term on the right of this equation represents a path of increasing length. For example, the third term is illustrated in Figure 15.5. This path has four vertices, connected by three segments. The total contribution of all such paths of length four (i.e., a vertex at the camera, two vertices at points on surfaces in the scene, and a vertex on a light source) is given by this term. Here, the first two vertices of the path, p_0 and p_1, are predetermined based on the camera ray and the point that it intersects, but p_2 and p_3 can vary over all points on surfaces in the scene. The integral over all such p_2 and p_3 gives the total contribution of paths of length four to radiance arriving at the camera.

This infinite sum can be written compactly as

$$L(p_1 \to p_0) = \sum_{n=1}^{\infty} P(\bar{p}_n). \tag{15.5}$$

$P(\bar{p}_n)$ gives the amount of radiance scattered over a path $\bar{p}_n$ with $n + 1$ vertices,

$$\bar{p}_n = p_0, p_1, \ldots, p_n,$$

where p_0 is on the film plane and p_n is on a light source, and

$$P(\bar{\mathrm{p}}_n) = \underbrace{\int_A \int_A \cdots \int_A}_{n-1} L_\mathrm{e}(\mathrm{p}_n \to \mathrm{p}_{n-1}) \times \left(\prod_{i=1}^{n-1} f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) G(\mathrm{p}_{i+1} \leftrightarrow \mathrm{p}_i) \right) \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n). \tag{15.6}$$

Before we move on, we will define one additional term that will be helpful in the subsequent discussion. The product of a path's BSDF and geometry terms is called the *throughput* of the path; it describes the fraction of radiance from the light source that arrives at the camera after all of the scattering at vertices between them. We will denote it by

$$T(\bar{\mathrm{p}}_n) = \prod_{i=1}^{n-1} f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) G(\mathrm{p}_{i+1} \leftrightarrow \mathrm{p}_i), \tag{15.7}$$

so

$$P(\bar{\mathrm{p}}_n) = \underbrace{\int_A \int_A \cdots \int_A}_{n-1} L_\mathrm{e}(\mathrm{p}_n \to \mathrm{p}_{n-1}) T(\bar{\mathrm{p}}_n) \, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n).$$

Given Equation (15.5) and a particular length n, all that we need to do to compute a Monte Carlo estimate of the radiance arriving at p_0 due to paths of length n is to sample a set of vertices with an appropriate sampling density in the scene to generate a path and then to evaluate an estimate of $P(\bar{\mathrm{p}}_n)$ using those vertices. Whether we generate those vertices by starting a path from the camera, starting from the light, starting from both ends, or starting from a point in the middle is a detail that only affects how the weights for the Monte Carlo estimates are computed. We will see how this formulation leads in practice to practical light transport algorithms throughout this chapter.

15.2.5 DELTA DISTRIBUTIONS IN THE INTEGRAND

Delta functions may be present in $P(\bar{\mathrm{p}}_i)$ terms due to both BSDF components described by delta distributions as well as certain types of light sources (e.g., point lights and directional lights). These distributions need to be handled explicitly by the light transport algorithm if present. For example, it is impossible to randomly choose an outgoing direction from a point on a surface that would intersect a point light source; instead, it is necessary to explicitly choose the single direction from the point to the light source if we want to be able to include its contribution. (The same is true for sampling BSDFs with delta components.) While handling this case introduces some additional complexity to the integrators, it is generally welcome because it reduces the dimensionality of the integral to be evaluated, turning parts of it into a plain sum.

For example, consider the direct illumination term, $P(\bar{\mathrm{p}}_2)$, in a scene with a single point light source at point $\mathrm{p}_{\mathrm{light}}$ described by a delta distribution:

$$\begin{aligned} P(\bar{\mathrm{p}}_2) &= \int_A L_{\mathrm{e}}(\mathrm{p}_2 \to \mathrm{p}_1)\, f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0)\, G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1)\, \mathrm{d}A(\mathrm{p}_2) \\ &= \frac{\delta(\mathrm{p}_{\mathrm{light}} - \mathrm{p}_2) L_{\mathrm{e}}(\mathrm{p}_{\mathrm{light}} \to \mathrm{p}_1)}{p(\mathrm{p}_{\mathrm{light}})} f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0) G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1). \end{aligned}$$

In other words, p_2 must be the same as the light's position in the scene; the delta distribution in the numerator cancels out due to an implicit delta distribution in $p(\mathrm{p}_{\mathrm{light}})$ (recall the discussion of sampling delta distributions in Section 14.5.4), and we are left with terms that can be evaluated directly, with no need for Monte Carlo. An analogous situation holds for BSDFs with delta distributions in the path throughput $T(\bar{\mathrm{p}}_n)$; each one eliminates an integral over area from the estimate to be computed.

15.2.6 PARTITIONING THE INTEGRAND

Many rendering algorithms have been developed that are particularly good at solving the LTE under some conditions, but don't work well (or at all) under others. For example, the Whitted integrator only handles specular reflection from delta BSDFs and ignores multiply scattered light from diffuse and glossy BSDFs, and the irradiance caching technique described later in this chapter handles scattering from diffuse surfaces but would introduce significant error if used for glossy or specular reflection.

Because we would like to be able to derive correct light transport algorithms that account for all possible modes of scattering without ignoring any contributions and without double-counting others, it is important to carefully account for which parts of the LTE a particular solution method accounts for. A nice way of approaching this problem is to partition the LTE in various ways. For example, we might expand the sum over paths to

$$L(\mathrm{p}_1 \to \mathrm{p}_0) = P(\bar{\mathrm{p}}_1) + P(\bar{\mathrm{p}}_2) + \sum_{i=3}^{\infty} P(\bar{\mathrm{p}}_i),$$

where the first term is trivially evaluated by computing the emitted radiance at p_1, the second term is solved with an accurate direct lighting solution technique, but the remaining terms in the sum are handled with a faster but less accurate approach. If the contribution of these additional terms to the total reflected radiance is relatively small for the scene we're rendering, this may be a reasonable approach to take. The only detail is that it is important to be careful to ignore $P(\bar{\mathrm{p}}_1)$ and $P(\bar{\mathrm{p}}_2)$ with the algorithm that handles $P(\bar{\mathrm{p}}_3)$ and beyond (and similarly with the other terms).

It is also useful to partition individual $P(\bar{\mathrm{p}}_n)$ terms. For example, we might want to split the emission term into emission from small light sources, $L_{\mathrm{e,s}}$, and emission from large light sources, $L_{\mathrm{e,l}}$, giving us two separate integrals to estimate:

$$\begin{aligned} P(\bar{\mathrm{p}}_n) &= \int_{A^{n-1}} (L_{\mathrm{e,s}}(\mathrm{p}_n \to \mathrm{p}_{n-1}) + L_{\mathrm{e,l}}(\mathrm{p}_n \to \mathrm{p}_{n-1}))\, T(\bar{\mathrm{p}}_n)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n) \\ &= \int_{A^n} L_{\mathrm{e,s}}(\mathrm{p}_n \to \mathrm{p}_{n-1})\, T(\bar{\mathrm{p}}_n)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n) \\ &\quad + \int_{A^n} L_{\mathrm{e,l}}(\mathrm{p}_n \to \mathrm{p}_{n-1})\, T(\bar{\mathrm{p}}_n)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n). \end{aligned}$$

The two integrals can be evaluated independently, possibly using completely different algorithms or different numbers of samples, selected in a way that handles the different conditions well. As long as the estimate of the $L_{\mathrm{e,s}}$ integral ignores any emission from large lights, the estimate of the $L_{\mathrm{e,l}}$ integral ignores emission from small lights, and all lights are categorized as either "large" or "small," the correct result is computed in the end.

Finally, the BSDF terms can be partitioned as well (in fact, this application was the reason why BSDF categorization with `BxDFType` values was introduced in Section 8.1). For example, if f_Δ denotes components of the BSDF described by delta distributions and $f_{\neg\Delta}$ denotes the remaining components,

$$\begin{aligned} P(\bar{\mathrm{p}}_n) = \int_{A^{n-1}} & L_{\mathrm{e}}(\mathrm{p}_n \to \mathrm{p}_{n-1}) \\ & \times \prod_{i=1}^{n-1} \left(f_\Delta(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) + f_{\neg\Delta}(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1}) \right) \\ & \qquad \times G(\mathrm{p}_{i+1} \leftrightarrow \mathrm{p}_i) \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_n). \end{aligned}$$

Note that because there are $i - 1$ BSDF terms in the product, it is important to be careful not to count only terms with only f_Δ components or only $f_{\neg\Delta}$ components; all of the terms like $f_\Delta f_{\neg\Delta} f_{\neg\Delta}$ must be accounted for as well if a partitioning scheme like this is used.

15.2.7 THE MEASUREMENT EQUATION AND IMPORTANCE

In light of the path integral form of the LTE, it's useful to go back and formally describe the quantity that is being estimated as we compute pixel values for the image. Doing so will help us be able to apply the LTE to a wider set of problems than just computing 2D images (for example, to precomputing scattered radiance distributions at the vertices of a polygonal model, giving a theoretical grounding for the precomputed light transport methods that will be introduced in Chapter 17). Furthermore, this process leads us to a key theoretical mechanism for understanding particle tracing and the photon mapping algorithm that will be described in Section 15.6.

The *measurement equation* describes the value of an abstract measurement that is found by integrating over some set of rays carrying radiance. For example, when computing the value of a pixel in the image, we want to integrate over rays starting in the neighborhood of the pixel, with contributions weighted by the image reconstruction filter. Ignoring depth of field for now (so that each point on the film plane corresponds to a single outgoing direction from the camera), we can write the pixel's value as an integral over points on the film plane of a weighting function times the incident radiance along the

corresponding camera rays:

$$\begin{aligned} I_j &= \int_{A_{\text{film}}} \int_{\mathcal{S}^2} W_{\text{e}}(\text{p}_{\text{film}}, \omega)\, L_{\text{i}}(\text{p}_{\text{film}}, \omega)\, |\cos\theta|\, \text{d}A(\text{p}_{\text{film}})\, \text{d}\omega \\ &= \int_{A_{\text{film}}} \int_{A} W_{\text{e}}(\text{p}_0 \rightarrow \text{p}_1)\, L(\text{p}_1 \rightarrow \text{p}_0)\, G(\text{p}_0 \leftrightarrow \text{p}_1)\, \text{d}A(\text{p}_0)\, \text{d}A(\text{p}_1), \end{aligned}$$

where I_j is the measurement for the jth pixel and p_0 is a point on the film. In this setting, the $W_{\text{e}}(\text{p}_0 \rightarrow \text{p}_1)$ term is the product of the filter function around the pixel, f_j, and a delta function that selects the appropriate camera ray direction of the sample from p_0, $\omega_{\text{camera}}(\text{p}_0)$:

$$W_{\text{e}}(\text{p}_0 \rightarrow \text{p}_1) = f_j(\text{p}_0)\, \delta(t(\text{p}_0, \omega_{\text{camera}}(\text{p}_0)) - \text{p}_1).$$

This formulation may initially seem gratuitously complex, but it leads us to an important insight. If we expand the $P(\bar{\text{p}}_n)$ terms of the LTE sum, we have

$$\begin{aligned} I_j &= \int_{A_{\text{film}}} \int_{A} W_{\text{e}}(\text{p}_0 \rightarrow \text{p}_1)\, L(\text{p}_1 \rightarrow \text{p}_0)\, G(\text{p}_0 \leftrightarrow \text{p}_1)\, \text{d}A(\text{p}_0)\, \text{d}A(\text{p}_1) \\ &= \sum_i \int_A \int_A W_{\text{e}}(\text{p}_0 \rightarrow \text{p}_1)\, P(\bar{\text{p}}_i)\, G(\text{p}_0 \leftrightarrow \text{p}_1)\, \text{d}A(\text{p}_0)\, \text{d}A(\text{p}_1) \\ &= \sum_i \int_A \cdots \int_A W_{\text{e}}(\text{p}_0 \rightarrow \text{p}_1)\, T(\bar{\text{p}}_i)\, L_{\text{e}}(\text{p}_{i+1} \rightarrow \text{p}_i)\, G(\text{p}_0 \leftrightarrow \text{p}_1)\, \text{d}A(\text{p}_0) \cdots \text{d}A(\text{p}_i). \end{aligned}$$

A nice symmetry between the emitted radiance from light sources (L_{e}) and the contribution of a sample on the film to the pixel measurement (W_{e}) has become apparent. The implications of this symmetry are important: it says that we can think of the rendering process in two different ways: Light could be emitted from light sources, bounce around the scene, and arrive at a sensor where W_{e} describes its contribution to the measurement. Alternatively, we can think of some quantity being emitted from the sensor, bouncing around the scene, and making a contribution when it hits a light source. Either intuition is equally valid.

The value described by the $W_{\text{e}}(\text{p}_0 \rightarrow \text{p}_1)$ term is known as the *importance* for the ray between p_0 and p_1 in the scene. When the measurement equation is used to compute pixel measurements, the importance will often be partially or fully described by delta distributions, as it was in the previous example. Many other types of measurement besides image formation can be described by appropriately constructed importance functions, and thus the formalisms described here can be used to show how the integral over paths described by the measurement equation is the integral that must be estimated to compute them. We will make use of these ideas when describing the photon mapping algorithm later in this chapter.

15.3 PATH TRACING

Now that we have derived the path integral form of the light transport equation, we'll show how it can be used to derive the *path-tracing* light transport algorithm and will present a path-tracing integrator. Figure 15.6 compares images of a scene rendered with



(a)



(b)

Figure 15.6: San Miguel Scene Rendered with Path Tracing. (a) Rendered with path tracing with 1024 samples per pixel. (b) Rendered with just 8 samples per pixel, giving the characteristic grainy noise that is the hallmark of variance. The disadvantage of path tracing is that a large number of samples may have to be taken in order to reduce variance to an acceptable level. *(Model courtesy of Guillermo M. Leal Llaguno.)*

different numbers of pixel samples using the path-tracing integrator. In general, hundreds or thousands of samples per pixel may be necessary for high-quality results—potentially a substantial computational expense.

Path tracing was the first general-purpose unbiased Monte Carlo light transport algorithm used in graphics. Kajiya (1986) introduced it in the same paper that first described the light transport equation. Path tracing incrementally generates paths of scattering events starting at the camera and ending at light sources in the scene. One way to think of it is as an extension of Whitted's method to include both delta distribution and nondelta BSDFs and light sources, rather than just accounting for the delta terms.

Although it is slightly easier to derive path tracing directly from the basic light transport equation, we will instead approach it from the path integral form, which helps build understanding of the path integral equation and will make the generalization to bidirectional path tracing easier to understand. Bidirectional path tracing is a technique where paths are generated starting from the lights as well as from the camera; it is discussed (but not implemented) at the end of this section. (See also Section 15.7.4, which has a simple bidirectional path-tracing implementation for the `MetropolisRenderer`.)

15.3.1 OVERVIEW

Given the path integral form of the LTE, we would like to estimate the value of the exitant radiance from the camera ray's intersection point p_1,

$$L(\mathrm{p}_1 \rightarrow \mathrm{p}_0) = \sum_{i=1}^{\infty} P(\bar{\mathrm{p}}_i),$$

for a given camera ray from p_0 that first intersects the scene at p_1. We have two problems that must be solved in order to compute this estimate:

1. How do we estimate the value of the sum of the infinite number of $P(\bar{\mathrm{p}}_i)$ terms with a finite amount of computation?
2. Given a particular $P(\bar{\mathrm{p}}_i)$ term, how do we generate one or more paths $\bar{\mathrm{p}}$ in order to compute a Monte Carlo estimate of its multidimensional integral?

For path tracing, we can take advantage of the fact that, for physically valid scenes, paths with more vertices scatter less light than paths with fewer vertices overall (this isn't necessarily true for any particular pair of paths, just in the aggregate). This is a natural consequence of conservation of energy in BSDFs. Therefore, we will always estimate the first few terms $P(\bar{\mathrm{p}}_i)$ and will then start to apply Russian roulette to stop sampling after a finite number of terms without introducing bias. Recall from Section 14.1 that Russian roulette allows us to probabilistically stop computing terms in a sum so long as we reweight the terms that are not terminated. For example, if we always computed estimates of $P(\bar{\mathrm{p}}_1)$, $P(\bar{\mathrm{p}}_2)$, and $P(\bar{\mathrm{p}}_3)$ but stopped without computing more terms with probability q, then an unbiased estimate of the sum would be

$$P(\bar{\mathrm{p}}_1) + P(\bar{\mathrm{p}}_2) + P(\bar{\mathrm{p}}_3) + \frac{1}{1-q} \sum_{i=4}^{\infty} P(\bar{\mathrm{p}}_i).$$

Using Russian roulette in this way doesn't solve the problem of needing to evaluate an infinite sum, but has pushed it a bit farther out.

If we take this idea a step further and instead randomly consider terminating evaluation of the sum at each term with probability q_i,

$$\frac{1}{1-q_1}\left(P(\bar{\mathrm{p}}_1) + \frac{1}{1-q_2}\left(P(\bar{\mathrm{p}}_2) + \frac{1}{1-q_3}\left(P(\bar{\mathrm{p}}_3) + \cdots,\right.\right.\right.$$

we will eventually stop continued evaluation of the sum. Yet, because for any particular value of i there is greater than zero probability of evaluating the term $P(\bar{\mathrm{p}}_i)$ and because it will be weighted appropriately if we do evaluate it, the final result is an unbiased estimate of the sum.

15.3.2 PATH SAMPLING

Given this method for evaluating only a finite number of terms of the infinite sum, we also need a way to estimate the contribution of a particular term $P(\bar{\mathrm{p}}_i)$. We need $i+1$ vertices to specify the path, where the last vertex p_i is on a light source and the first vertex p_0 is determined by the camera ray's first intersection point (Figure 15.7). Looking at the form of $P(\bar{\mathrm{p}}_i)$, a multiple integral over surface area of objects in the scene, the most natural thing to do is to sample vertices p_i according to the surface area of objects in the scene, such that it's equally probable to sample any particular point on an object in the scene for p_i as any other point. (We don't actually use this approach in the `PathIntegrator` implementation for reasons that will be described later, but this sampling technique could possibly be used to improve the efficiency of our basic implementation and helps to clarify the meaning of the path integral LTE.)

We could define a discrete probability over the n objects in the scene. If each has surface area A_i, then the probability of sampling a path vertex on the surface of the ith object

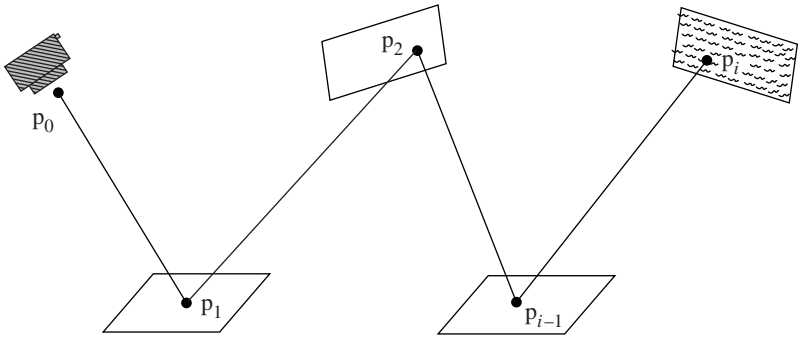


Figure 15.7: A path $\bar{\mathrm{p}}_i$ of $i+1$ vertices from the camera at p, intersecting a series of positions on surfaces in the scene, to a point on the light p_i. Scattering according to the BSDF occurs at each path vertex from p_i to p_{i-1} such that the radiance arriving at the camera due to this path is given by the product of the path throughput $T(\bar{\mathrm{p}}_i)$ times the emitted radiance from the light.

should be

$$p_i = \frac{A_i}{\sum_j A_j}.$$

Then, given a method to sample a point on the ith object with uniform probability, the PDF for sampling any particular point on object i is $1/A_i$. Thus, the overall probability density for sampling the point is

$$\frac{A_i}{\sum_j A_j}\frac{1}{A_i}.$$

And all samples p_i have the same PDF value:

$$p_A(\mathrm{p}_i) = \frac{1}{\sum_j A_j}.$$

It's reassuring that they all have the same weight, since our intent was to choose among all points on surfaces in the scene with equal probability.

Given the set of vertices $\mathrm{p}_0, \mathrm{p}_1, \ldots, \mathrm{p}_{i-1}$ sampled in this manner, we can then sample the last vertex p_i on a light source in the scene, defining its PDF in the same way. Although we could use the same technique used for sampling path vertices to sample points on lights, this would lead to high variance, since for all of the paths where p_i wasn't on the surface of an emitter, the path would have zero value. The expected value would still be the correct value of the integral, but convergence would be extremely slow. A better approach is to sample over the areas of only the emitting objects with probabilities updated accordingly. Given a complete path, we have all of the information we need to compute the estimate of $P(\bar{\mathrm{p}}_i)$; it's just a matter of evaluating each of the terms.

It's easy to be more creative about how we set the sampling probabilities with this general approach. For example, if we knew that indirect illumination from a few objects contributed to most of the lighting in the scene, we could assign a higher probability to generating path vertices p_i on those objects, updating the sample weights appropriately.

However, there are two interrelated problems with sampling paths in this manner. The first can lead to high variance, while the second can lead to incorrect results. The first problem is that many of the paths will have no contribution if they have pairs of adjacent vertices that are not mutually visible. Consider applying this area sampling method in a complex building model: adjacent vertices in the path will almost always have a wall or two between them, giving no contribution for the path and excessive variance in the estimate.

The second problem is that if the integrand has delta functions in it (e.g., a point light source or a perfectly specular BSDF), this sampling technique will never be able to choose path vertices such that the delta distributions are nonzero. And, even if there aren't delta distributions, as the BSDFs become increasingly glossy almost all of the paths will have low contributions since the points in $f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1})$ will cause the BSDF to have a small or zero value and again we will suffer from high variance. In a similar manner, small area light sources can also be sources of variance if not sampled explicitly.

15.3.3 INCREMENTAL PATH CONSTRUCTION

A solution that solves both of these problems is to construct the path incrementally, starting from the vertex at the camera p_0. At each vertex, the BSDF is sampled to generate a new direction; the next vertex p_{i+1} is found by tracing a ray from p_i in the sampled direction and finding the closest intersection. We are effectively trying to find a path with a large overall contribution by making a series of choices that find directions with important local contributions. While one can imagine situations where this approach could be ineffective, it is generally a good strategy.

Because this approach constructs the path by sampling BSDFs according to solid angle, and because the path integral LTE is an integral over surface area in the scene, we need to apply the correction to convert from the probability density according to solid angle p_ω to a density according to area p_A (recall Section 5.5):

$$p_A = p_\omega \frac{|\cos \theta_i|}{\|\mathrm{p}_i - \mathrm{p}_{i+1}\|^2}.$$

This correction causes some of the terms of the geometric term $G(\mathrm{p}_i \leftrightarrow \mathrm{p}_{i+1})$ to cancel out of $P(\bar{\mathrm{p}}_i)$. Furthermore, we already know that p_i and p_{i+1} must be mutually visible since we traced a ray to find p_{i+1}, so the visibility term is trivially one. Therefore, if we use this sampling technique but we still sample the last vertex p_i from some distribution over the surfaces of light sources $p_A(\mathrm{p}_i)$, the value of the Monte Carlo estimate for a path is

$$\frac{L_\mathrm{e}(\mathrm{p}_i \to \mathrm{p}_{i-1}) f(\mathrm{p}_i \to \mathrm{p}_{i-1} \to \mathrm{p}_{i-2}) |\cos \theta_{\mathrm{i}-1}|}{p_A(\mathrm{p}_i)} \left(\prod_{j=1}^{i-2} \frac{f(\mathrm{p}_{j+1} \to \mathrm{p}_j \to \mathrm{p}_{j-1}) |\cos \theta_j|}{p_\omega(\mathrm{p}_{j+1} - \mathrm{p}_j)} \right). \tag{15.8}$$

15.3.4 IMPLEMENTATION

Our path-tracing implementation computes an estimate of the sum of path contributions $P(\bar{\mathrm{p}}_i)$ using the approach described in the previous subsection. Starting at the first intersection of the camera ray with the scene geometry, p_1, it incrementally samples path vertices by sampling from the BSDF's sampling distribution at the current vertex and tracing a ray to the next vertex. To find the last vertex of a particular path, p_i, which must be on a light source in the scene, it uses the multiple importance sampling–based direct lighting code that was developed for the direct lighting integrator. By using the multiple importance sampling weights instead of $p_A(\mathrm{p}_i)$ to compute the estimate as described earlier, we have lower variance in the result for cases where sampling the BSDF would have been a better way to find a point on the light.

Another small difference is that as the estimates of the path contribution terms $P(\bar{\mathrm{p}}_i)$ are being evaluated, the vertices of the previous path of length $i - 1$ (everything except the vertex on the emitter) are reused as a starting point when constructing the path of length i. This means that it is only necessary to trace one more ray to construct the new path, rather than i rays as we would if we started from scratch. Reusing paths in this manner does introduce correlation among all of the $P(\bar{\mathrm{p}}_i)$ terms in the sum, which slightly reduces the quality of the result, although in practice this is more than made up for by the improved overall efficiency due to tracing fewer rays.

⟨*PathIntegrator Declarations*⟩ ≡
```
class PathIntegrator : public SurfaceIntegrator {
public:
    ⟨PathIntegrator Public Methods 766⟩
private:
    ⟨PathIntegrator Private Data 766⟩
};
```

Although Russian roulette is used here to terminate path sampling in the manner described earlier, the integrator also supports a maximum depth. It can be set to a large number if only Russian roulette should be used:

⟨*PathIntegrator Public Methods*⟩ ≡ 766
```
PathIntegrator(int md) { maxDepth = md; }
```

⟨*PathIntegrator Private Data*⟩ ≡ 766
```
int maxDepth;
```

The integrator uses samples from the `Sampler` for sampling at the first `SAMPLE_DEPTH` vertices of the path. After the first few bounces, the advantages of well-distributed sample points are greatly reduced, and it switches to using uniform random numbers. The integrator needs light and BSDF samples for multiple importance sampling for the direct lighting calculation at each vertex of the path as well as a second set of BSDF samples for sampling directions when generating the outgoing direction for finding the next vertex of the path.

⟨*PathIntegrator Method Definitions*⟩ ≡
```
void PathIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
                                    const Scene *scene) {
    for (int i = 0; i < SAMPLE_DEPTH; ++i) {
        lightSampleOffsets[i] = LightSampleOffsets(1, sample);
        lightNumOffset[i] = sample->Add1D(1);
        bsdfSampleOffsets[i] = BSDFSampleOffsets(1, sample);
        pathSampleOffsets[i] = BSDFSampleOffsets(1, sample);
    }
}
```

⟨*PathIntegrator Private Data*⟩ +≡ 766
```
#define SAMPLE_DEPTH 3
LightSampleOffsets lightSampleOffsets[SAMPLE_DEPTH];
int lightNumOffset[SAMPLE_DEPTH];
BSDFSampleOffsets bsdfSampleOffsets[SAMPLE_DEPTH];
BSDFSampleOffsets pathSampleOffsets[SAMPLE_DEPTH];
```

Each time through the `for` loop of the integrator, the next vertex of the path is found by intersecting the current ray with the scene geometry and computing the contribution of the path to the overall radiance value with the direct lighting code. A new direction is then chosen by sampling from the BSDF's distribution at the last vertex of the path. After a few vertices have been sampled, Russian roulette is used to randomly terminate the path.

BSDFSampleOffsets 706
LightSampleOffsets 710
PathIntegrator 766
PathIntegrator::
bsdfSampleOffsets 766
PathIntegrator::
lightNumOffset 766
PathIntegrator::
lightSampleOffsets 766
PathIntegrator::
pathSampleOffsets 766
Sample 343
Sample::Add1D() 344
Sampler 340
Scene 22
SurfaceIntegrator 740

⟨*PathIntegrator Method Definitions*⟩ +≡
```
Spectrum PathIntegrator::Li(const Scene *scene, const Renderer *renderer,
        const RayDifferential &r, const Intersection &isect,
        const Sample *sample, RNG &rng, MemoryArena &arena) const {
    ⟨Declare common path integration variables 767⟩
    for (int bounces = 0; ; ++bounces) {
        ⟨Possibly add emitted light at path vertex 768⟩
        ⟨Sample illumination from lights to find path contribution 768⟩
        ⟨Sample BSDF to get new path direction 768⟩
        ⟨Possibly terminate the path 769⟩
        ⟨Find next vertex of path 769⟩
    }
    return L;
}
```

A number of variables record the current state of the path. `pathThroughput` holds the product of the BSDF values and cosine terms for the vertices generated so far, divided by their respective sampling PDFs,

$$\prod_{j=1}^{i-2} \frac{f(\mathrm{p}_{j+1} \to \mathrm{p}_j \to \mathrm{p}_{j-1})|\cos\theta_j|}{p_\omega(\mathrm{p}_{j+1} - \mathrm{p}_j)}.$$

Thus, the product of `pathThroughput` with scattered light from direct lighting at the final vertex of the path gives the contribution for that overall path. One advantage of this approach is that there is no need to store the positions and BSDFs of all of the vertices of the path, only the last one.

`L` holds the radiance value from the running total of $\sum P(\bar{\mathrm{p}}_i)$, `ray` holds the next ray to be traced to extend the path one more vertex, and `specularBounce` records if the last outgoing path direction sampled was due to specular reflection; the need to track this will be explained shortly.

Finally, `isectp` points to the `Intersection` for the most recently added vertex in the path. The first time through the `for` loop, it points to `isect`, which was passed in to the `Li()` method for the first intersection of the ray with scene geometry. Information about subsequent intersections is stored in `localIsect`; `isectp` is updated to point to this after the first intersection is processed.

⟨*Declare common path integration variables*⟩ ≡ 767
```
Spectrum pathThroughput = 1., L = 0.;
RayDifferential ray(r);
bool specularBounce = false;
Intersection localIsect;
const Intersection *isectp = &isect;
```

Intersection 186
MemoryArena 1015
PathIntegrator 766
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Scene 22
Spectrum 263

If the ray hits an object that is emissive, the emission is usually ignored, since the previous path vertex already did a direct lighting computation that was responsible for all direct lighting for paths of this length. There are two exceptions to this: The first is at the initial intersection point of camera rays, since this is the only opportunity to include emission from directly visible objects. The second exception happens when the sampled direction

from the last path vertex was from a specular BSDF component. EstimateDirect() deliberately omits the effect of specular reflection in the direct lighting computation with its default parameters; therefore, if the last bounce was due to specular reflection, any emission at the intersection point must be included here.

⟨*Possibly add emitted light at path vertex*⟩ ≡ 767
```
if (bounces == 0 || specularBounce)
    L += pathThroughput * isectp->Le(-ray.d);
```

The direct lighting computation uses the UniformSampleOneLight() function, which gives an estimate of the exitant radiance from direct lighting at the vertex at the end of the current path. Scaling this value by the running product of the path contribution gives its overall contribution to the total radiance estimate.

⟨*Sample illumination from lights to find path contribution*⟩ ≡ 767
```
BSDF *bsdf = isectp->GetBSDF(ray, arena);
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Vector wo = -ray.d;
if (bounces < SAMPLE_DEPTH)
    L += pathThroughput *
         UniformSampleOneLight(scene, renderer, arena, p, n, wo,
             isectp->rayEpsilon, ray.time, bsdf, sample, rng,
             lightNumOffset[bounces], &lightSampleOffsets[bounces],
             &bsdfSampleOffsets[bounces]);
else
    L += pathThroughput *
         UniformSampleOneLight(scene, renderer, arena, p, n, wo,
             isectp->rayEpsilon, ray.time, bsdf, sample, rng);
```

Now it is necessary to sample the BSDF at the vertex at the end of the current path to get an outgoing direction for the next ray to trace. The ⟨*Get* outgoingBSDFSample *for sampling new path direction*⟩ fragment initializes a BSDFSample from sample if the current path length is less than the value of SAMPLE_DEPTH, or uses RNG::RandomFloat() otherwise. The integrator updates the path throughput as described earlier and initializes ray with the ray to be traced to find the next vertex in the next iteration of the for loop.

⟨*Sample BSDF to get new path direction*⟩ ≡ 767
```
⟨Get outgoingBSDFSample for sampling new path direction 769⟩
Vector wi;
float pdf;
BxDFType flags;
Spectrum f = bsdf->Sample_f(wo, &wi, outgoingBSDFSample, &pdf,
                            BSDF_ALL, &flags);
if (f.IsBlack() || pdf == 0.)
    break;
specularBounce = (flags & BSDF_SPECULAR) != 0;
pathThroughput *= f * AbsDot(wi, n) / pdf;
ray = RayDifferential(p, wi, ray, isectp->rayEpsilon);
```

AbsDot() 61
BSDF 478
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
EstimateDirect() 749
Intersection::GetBSDF() 484
Intersection::Le() 625
Normal 65
PathIntegrator::
bsdfSampleOffsets 766
PathIntegrator::
lightNumOffset 766
PathIntegrator::
lightSampleOffsets 766
Point 63
Ray::d 67
RayDifferential 69
RNG::RandomFloat() 1003
Spectrum 263
Spectrum::IsBlack() 265
UniformSampleOneLight() 746
Vector 57

⟨*Get* `outgoingBSDFSample` *for sampling new path direction*⟩ ≡ 768
```
BSDFSample outgoingBSDFSample;
if (bounces < SAMPLE_DEPTH)
    outgoingBSDFSample = BSDFSample(sample, pathSampleOffsets[bounces],
                                    0);
else
    outgoingBSDFSample = BSDFSample(rng);
```

Path termination kicks in after a few bounces, with termination probability set with a fixed minimum or based on the path throughput, whichever is lower. In general, it's worth having a higher probability of terminating low-contributing paths, since they have relatively less impact on the final image. If the path isn't terminated, `pathThroughput` is updated with the Russian roulette weight and all subsequent $P(\bar{\mathrm{p}}_i)$ terms will be appropriately affected by it.

⟨*Possibly terminate the path*⟩ ≡ 767
```
if (bounces > 3) {
    float continueProbability = min(.5f, pathThroughput.y());
    if (rng.RandomFloat() > continueProbability)
        break;
    pathThroughput /= continueProbability;
}
if (bounces == maxDepth)
    break;
```

The last task in the loop is to find the next vertex of the path by tracing a ray in the sampled outgoing direction. If no intersection is found, the ray has escaped the scene and the path is terminated; if the sample from the last path vertex was due to specular reflection or transmission, then the incident illumination from light sources that aren't represented by geometry in the scene must be included here. No direct lighting computation is done when a specular BSDF sample is taken, so if the ray misses scene geometry, we still need to account for infinite area light sources—if their contribution wasn't added here, infinite area light sources wouldn't be visible in surfaces with perfect specular reflection.

In the usual case where the ray does intersect scene geometry, the path throughput is updated to account for attenuation along the new segment due to participating media and the `isectp` pointer is updated to ensure that it points at the `Intersection` for the geometric intersection just found.

⟨*Find next vertex of path*⟩ ≡ 767
```
if (!scene->Intersect(ray, &localIsect)) {
    if (specularBounce)
        for (uint32_t i = 0; i < scene->lights.size(); ++i)
            L += pathThroughput * scene->lights[i]->Le(ray);
    break;
}
if (bounces > 1)
    pathThroughput *= renderer->Transmittance(scene, ray, NULL, rng, arena);
isectp = &localIsect;
```

BSDFSample 705
Intersection 186
Light::Le() 631
PathIntegrator::
pathSampleOffsets 766
Renderer::Transmittance() 25
RNG::RandomFloat() 1003
Scene::Intersect() 23
Scene::lights 23
Spectrum::y() 273

★ 15.3.5 BIDIRECTIONAL PATH TRACING

The path-tracing algorithm described in this section was the first general light transport algorithm in graphics, handling both a wide variety of geometric objects as well as area lights and general BSDF models. Although it works well for many scenes, it can exhibit high variance in the presence of particular tricky lighting conditions. For example, consider the setting shown in Figure 15.8: a light source is illuminating a small area on the ceiling such that the rest of the room is only illuminated by indirect lighting bouncing from that area. If we only trace paths starting from the camera, we will almost never happen to sample a path vertex in the illuminated region on the ceiling before we trace a shadow ray to the light. Most of the paths will have no contribution, while a few of them—the ones that happen to hit the small region on the ceiling—will have a large contribution. The resulting image will have high variance.

Difficult lighting settings like this can be handled more effectively by constructing paths that start from the camera on one end and from the light on the other end and are connected in the middle with a visibility ray. This *bidirectional path-tracing* algorithm is a generalization of the standard path-tracing algorithm; for the same amount of computation, it can give substantially lower variance.

The path integral form of the LTE makes it easy to understand how to construct a bidirectional algorithm. As with standard path tracing, the first vertex, p_1, is found by computing the first intersection along the camera ray. The last vertex is found by sampling a point on a light source in the scene. Here, we will label the last vertex as q_1, so that we can construct a path of not initially determined length "backward" from the light.

In the basic bidirectional path-tracing algorithm, we go forward from the camera to create a subpath $p_1, p_2, \ldots, p_i$ and backward from the light to compute a subpath

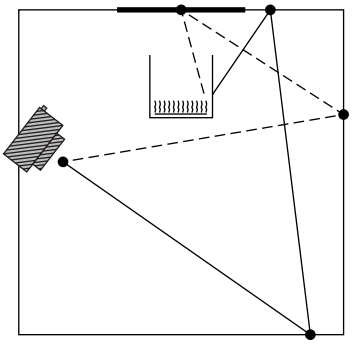


Figure 15.8: A Difficult Case for Path Tracing Starting from the Camera. A light source is illuminating a small area on the ceiling such that only paths with a second-to-last vertex in the area indicated will be able to find illumination from the light. Bidirectional methods, where a path is started from the light and is connected with a path from the camera, can handle situations like these more robustly.

$q_1, q_2, \ldots, q_j$. Each subpath is usually computed incrementally by sampling the BSDF at the previous vertex, although other sampling approaches can be used in the same way as was described for standard path tracing. (Weights for each vertex are computed in the same manner as well.) In either case, in the end, we have a path

$$\bar{p} = p_1, \ldots, p_i, q_j, \ldots, q_1.$$

We need to trace a shadow ray between p_i and q_j to see if they are mutually visible; if so, the path carries light from the light to the camera, and we can evaluate the path's contribution directly.

There are three refinements to this basic algorithm that improve its performance in practice. The first two are analogous to improvements made to path tracing, and the third is a powerful variance reduction technique on top of the improvement from just starting at the light as well.

- First, subpaths can be reused: given a path $p_1, \ldots, p_i, q_j, \ldots, q_1$, transport over all of the paths can be evaluated by connecting all the various combinations of prefixes of the two paths together. If the two paths have i and j vertices, respectively, then a variety of unique paths can be constructed from them, ranging in length from 2 to $i + j$ vertices long. Each such path built this way requires only that a visibility check be performed by tracing a shadow ray between the last vertices of each of the subpaths. (The `BSDF`s for each vertex of both paths must be stored to do this.)
- The second optimization is to ignore the paths generated in the path reuse stage that only use one vertex from the light subpath and instead to use the optimized direct lighting code from the direct lighting integrator. This gives a lower-variance result than using the vertex on the light sampled for the light subpath, since it makes it possible both to use multiple importance sampling with the BSDF and to use well-distributed sampling patterns for this part of the problem.
- The third optimization is to use multiple importance sampling to reweight paths. Recall the example of a light pointed up at the ceiling, indirectly illuminating a room. As described so far, bidirectional path tracing will improve the result substantially by greatly reducing the number of paths with no contribution, since the paths from the light will be effective at finding those light transport routes. However, the image will still suffer from variance due to paths with unexpectedly large contributions—for example, from paths from the camera that happened to find the bright spot in the ceiling. MIS can be applied to solve this, recognizing that for a path with n vertices, there are actually $n - 1$ ways a path with that length could be generated. For example, a four-vertex path could be built from one camera vertex and three light vertices, two of each kind of vertex, or three camera vertices and one light vertex. Given a particular path sampled in a particular way, we can compute the weights for each of the other ways the path could have been generated and apply the balance heuristic.

BSDF 478
MetropolisRenderer 852

`pbrt` doesn't have a bidirectional path-tracing integrator; implementing one is left as an exercise. However, bidirectional path tracing is used in the `MetropolisRenderer` implemented in Section 15.7.



(a) (b) (c)

Figure 15.9: Indoor Scene Rendered with "Instant Global Illumination." (a) Scene rendered with direct illumination only. (b) Using just four virtual light sources with the `IGIIntegrator`. There is indirect lighting, but because so few virtual lights were used, their shadows are distracting. (c) Using 64 virtual lights with the `IGIIntegrator` gives a high-quality result. *(Model courtesy of Florent Boyer.)*

15.4 INSTANT GLOBAL ILLUMINATION

The path-tracing integrator computes unbiased estimates of the scene radiance, though at a cost of requiring from tens to thousands of samples in each pixel in order to compute an image that isn't noisy. This computational cost can be undesirable; this and the following sections will introduce more efficient approaches to solving the LTE, some of them doing so by more effectively reusing intermediate results, sometimes introducing bias.

This section describes the "instant global illumination" (IGI) integrator. The basic idea is to follow a small number of light-carrying paths from the light sources and to construct a number of point light sources at the points where these paths intersect surfaces in the scene. These point sources (also sometimes called *virtual light sources*) are deposited in a way such that their illumination approximates the indirect radiance distribution in the scene. After these lights have been created, when the integrator later needs to compute exitant radiance at a point, it loops over these point light sources and traces shadow rays to see if they are visible, accumulating the indirect illumination that they represent if so. This approach to global illumination has started to see some use in interactive rendering; the "Further Reading" section at the end of this chapter has pointers to relevant papers.

Figure 15.9 shows images of the contemporary house scene rendered with direct lighting and with IGI with different parameter settings. IGI renders indirect lighting effects very efficiently, roughly doubling in computation time for Figure 15.9(c) versus Figure 15.9(a), and without the noise characteristic of algorithms like path tracing or bidirectional path tracing. It can be implemented as an unbiased light transport algorithm; however, because the same virtual lights are used for all of the pixels in the scene, error shows up as systemic error due to correlation as opposed to noise as would be the case for path-tracing algorithms. While this is often an acceptable property for single images, this error can be distracting in animations, where subsequent frames may have noticeably different brightnesses if a different set of lights is generated for each one.

IGIIntegrator 773

In this section, we will show how IGI can be understood as a variant of bidirectional path tracing (see Section 15.3.5). This formulation both makes it easier to understand which cosine terms and such to include in the expressions computed and also makes it easier to see how the approach could be modified. However, while we will use this theory as the foundation for the derivation of the approach, the variable names and such in the implementation will still be in terms of the "virtual light source" approach. The `IGIIntegrator` is implemented in the files `integrators/igi.h` and `integrators/igi.cpp`. As usual, the standard parts of the implementation of the integrator will be elided here, and we will just focus on the parts that are unique to IGI.

The `IGIIntegrator` constructor takes seven parameters. Its implementation is not included here, as it just sets the following member variables from the parameters given.

The first parameter, `nLightPaths`, gives the number of paths to follow from lights to create the virtual lights (the actual number of virtual lights created will generally be larger than this, as each path usually creates multiple lights).

The second parameter, `nLightSets`, describes the number of light sets to create; instead of computing just one set of virtual lights, this integrator actually computes `nLightSets` independent sets of them. To see why doing so is useful, consider an image being rendered at a rate of 16 image samples per pixel: in general, we will expect a better result if each image sample uses a different set of virtual lights, thus incorporating more information about indirect illumination in the scene at little incremental computational cost. Thus, for best results, this value should be the same as the number of samples taken per pixel. [3]

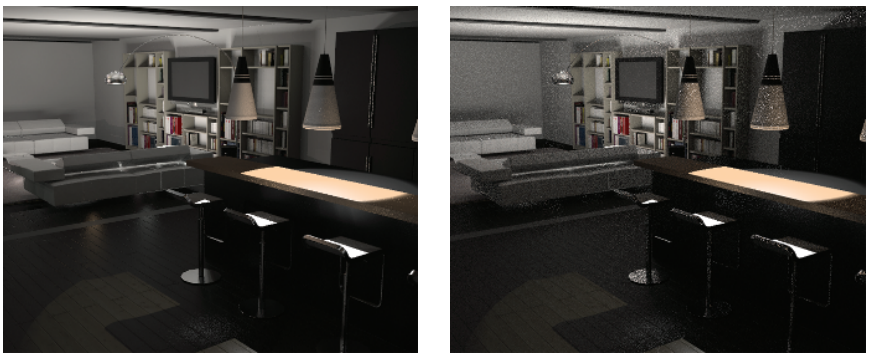
⟨*IGIIntegrator Private Data*⟩ ≡
```
uint32_t nLightPaths, nLightSets;
```

Figure 15.10 shows the example scene rendered again with just one set of lights (a), and with 256 sets (b), both images rendered with 16 samples per pixel. In both cases, image quality suffers. With just one set, all 16 image samples use the same set of virtual lights. As this set only has 10 or so lights in it, indirect lighting cannot be represented accurately, and the virtual lights cast noticeable shadows. With 256 light sets but just 16 pixel samples, as in the second image, the result is noisy, as different pixels will generally use a different set of lights. This case is an interesting illustration of the fact that minimizing numeric error doesn't necessarily minimize the error perceived by a human viewer: increasing numbers of light sets will in general reduce the systemic error due to correlation from re-using the same light set many times. However, this error reduction introduces noise, which is more obviously objectionable to a human observer.

Part of the IGI lighting computation involves evaluating an expression that includes the $G(\mathrm{p} \leftrightarrow \mathrm{p}')$ term that was defined in Equation (15.3). This term can exhibit an *weak singularity* when the distance between p and p′ is small, such that one over the squared distance between them gives a very large number. To avoid bright image artifacts when

IGIIntegrator 773

3 The integrator depends on a well-written `Sampler` to generate a stratified set of samples for the usual case where the number of pixel samples is the same as the number of virtual light sets. Otherwise, the one-to-one relationship between the two may be lost.



(a) (b)

Figure 15.10: Varying the Number of Light Sets with Instant Global Illumination. Scene rendered with 16 samples per pixel. (a) If just one light set is generated, all 16 pixel samples use the same set of virtual lights and aren't able to incorporate as much information about the illumination as if they each used different sets. (b) However, with 256 light sets, the pixel samples in each pixel can use only a subset of them, so that different pixels use different sets for their lighting, introducing noise to the final result. *(Model courtesy of Florent Boyer.)*

this is the case, a different sampling strategy is used whenever the G term is greater than `gLimit`, taking `nGatherSamples` in that case.

⟨*IGIIntegrator Private Data*⟩ +≡
```
float gLimit;
int nGatherSamples;
```

Finally, `rrThreshold` is a Russian roulette threshold used to skip tracing rays to virtual lights that make a small contribution to a particular point where exitant radiance is being computed, and `maxSpecularDepth` is used to terminate specular reflection after a few bounces in the usual manner.

⟨*IGIIntegrator Private Data*⟩ +≡
```
float rrThreshold;
int maxSpecularDepth;
```

In the integrator's `Li()` method, the typical sampling approach is used for the direct lighting computation, where samples are taken from all light sources. In addition to the usual samples needed for that computation, this integrator also needs a well-distributed 1D sample to choose among the multiple sets of virtual lights and may need samples for the gathering sampling algorithm in the weak singularity case; `RequestSamples()` handles notifying the `Sampler` of all of these needed sets of samples.

⟨*IGIIntegrator Method Definitions*⟩ ≡
```
void IGIIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
                                   const Scene *scene) {
    ⟨Allocate and request samples for sampling all lights 743⟩
    vlSetOffset = sample->Add1D(1);
```

BSDFSampleOffsets 706
IGIIntegrator 773
IGIIntegrator::
gatherSampleOffset 775
IGIIntegrator::
nGatherSamples 774
IGIIntegrator::
vlSetOffset 775
Sample 343
Sample::Add1D() 344
Sampler 340
Scene 22

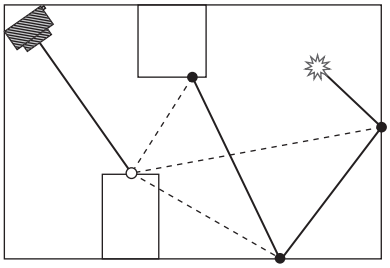


Figure 15.11: Connecting Paths with the `IGIIntegrator`. With the "instant global illumination" algorithm, a number of light-carrying paths are constructed from the light sources; the virtual lights created by such a path are indicated with filled circles here. When a point is being shaded (open circle), shadow rays (dashed lines) are traced between the point and the vertices of the light path to compute indirect illumination. Shadow rays to the light source itself are handled separately, using the direct lighting techniques from Section 15.1.

```
    if (sampler) nGatherSamples = sampler->RoundSize(nGatherSamples);
    gatherSampleOffset = BSDFSampleOffsets(nGatherSamples, sample);
}
```

⟨*IGIIntegrator Private Data*⟩ +≡

```
int vlSetOffset;
BSDFSampleOffsets gatherSampleOffset;
```

15.4.1 CREATING THE VIRTUAL LIGHT SOURCES

The task of the `Preprocess()` routine is to follow light-carrying paths from the light sources and to create the virtual point sources. To understand which values it needs to precompute and store with each virtual light, recall from the discussion of bidirectional path tracing in Section 15.3.5 that we can write the exitant radiance from a point p_1 to a point on the film plane p_0 as an infinite sum over light-carrying paths where each path is an integral over points on the surfaces of objects in the scene. IGI corresponds to the case of bidirectional path tracing where the path from the camera has only one segment and the light path has one or more segments.[4] Figure 15.11 shows an example where the light path has three vertices and has been connected to a camera path of one segment with shadow rays to the vertices of the light path (dotted line). The camera segment and the connecting segment are uniquely determined given the camera ray, its intersection point, and the vertex of the light path it is being connected to.

In this approach, there is only one way to generate any path of length n segments: one camera segment from p_0 to p_1, one connecting segment from p_1 to p_2, and $n - 2$ light segments from p_2 to p_n. Applying the Monte Carlo estimator, the Monte Carlo estimate

BSDFSampleOffsets 706

4 Note that direct lighting and specular reflection thus aren't accounted for here but are easily handled separately with conventional approaches.

of a path's contribution, Equation (15.6), can be written for this case as

$$P(\bar{\mathrm{p}}_n) = \alpha\, f(\mathrm{p}_3 \to \mathrm{p}_2 \to \mathrm{p}_1) G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1) f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0), \qquad [15.9]$$

where

$$\alpha = \frac{L_{\mathrm{e}}(\mathrm{p}_n \to \mathrm{p}_{n-1}) f(\mathrm{p}_n \to \mathrm{p}_{n-1} \to \mathrm{p}_{n-2}) |\cos\theta_{\mathrm{n}-1}|}{p_A(\mathrm{p}_n)} \times \left(\prod_{i=3}^{n-2} \frac{f(\mathrm{p}_{i+1} \to \mathrm{p}_i \to \mathrm{p}_{i-1})\ |\cos\theta_i|}{p_\omega(\mathrm{p}_{i+1} - \mathrm{p}_i)} \right).$$

The α term is independent of both the segment from the camera to the first visible point p_1 as well as the segment from that point to the last vertex of the light path p_2. Therefore, if we precompute this term, then computing the effect of the corresponding virtual light just requires being able to evaluate the two remaining BSDF terms and the geometric term $G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1)$.

The `Preprocess()` method creates `nLightSets` sets of paths from the lights in the scene, where each set has `nLightPaths` paths from lights. Each such light path will generally lead to multiple virtual light sources, as one virtual light is created for each vertex in the path after the one on the light source. (Russian roulette is used to terminate paths in an unbiased manner.) Because this step usually requires tracing only a small number of rays, we haven't parallelized it by decomposing it into tasks.

⟨*IGIIntegrator Method Definitions*⟩ +≡

```
void IGIIntegrator::Preprocess(const Scene *scene, const Camera *camera,
                               const Renderer *renderer) {
    if (scene->lights.size() == 0) return;
    MemoryArena arena;
    RNG rng;
    ⟨Compute samples for emitted rays from lights 777⟩
    ⟨Precompute information for light sampling densities 777⟩
    for (uint32_t s = 0; s < nLightSets; ++s) {
        for (uint32_t i = 0; i < nLightPaths; ++i) {
            ⟨Follow path i from light to create virtual lights 777⟩
        }
    }
}
```

To ensure a good sampling of indirect light, it's helpful to use well-distributed samples to choose the initial points on the light sources and their directions. The `LDShuffleScrambled*D()` functions work well to compute samples for generating the initial rays from the lights. Using them in this manner ensures not only that the samples used for each particular set of virtual lights are well distributed but also that in the aggregate over all of the paths the samples are globally well distributed as well.

Camera 302
IGIIntegrator 773
MemoryArena 1015
Renderer 24
RNG 1003
Scene 22

⟨*Compute samples for emitted rays from lights*⟩ ≡ 776
```
vector<float> lightNum(nLightPaths * nLightSets);
vector<float> lightSampPos(2 * nLightPaths * nLightSets, 0.f);
vector<float> lightSampComp(nLightPaths * nLightSets, 0.f);
vector<float> lightSampDir(2 * nLightPaths * nLightSets, 0.f);
LDShuffleScrambled1D(nLightPaths, nLightSets, &lightNum[0], rng);
LDShuffleScrambled2D(nLightPaths, nLightSets, &lightSampPos[0], rng);
LDShuffleScrambled1D(nLightPaths, nLightSets, &lightSampComp[0], rng);
LDShuffleScrambled2D(nLightPaths, nLightSets, &lightSampDir[0], rng);
```

The efficiency of the algorithm is also improved if the probability of starting a path from a light source is related to the amount of power it emits in comparison to the power emitted by the other lights. (The intensity of the corresponding virtual lights must be reduced appropriately to counterbalance the fact that more virtual lights are created for bright light sources.) Increasing the probability of generating paths from bright lights naturally leads to more virtual lights being created for those lights. The `ComputeLightSamplingCDF()` routine computes a discrete probability density for sampling each light according to its power, returning a `Distribution1D` to represent the distribution.

⟨*Precompute information for light sampling densities*⟩ ≡ 776
```
Distribution1D *lightDistribution = ComputeLightSamplingCDF(scene);
```

After a light source is chosen and an initial ray leaving it is sampled, the path is followed, with virtual lights stored at each vertex, until the ray leaves the scene or the path is terminated with Russian roulette.

⟨*Follow path* `i` *from light to create virtual lights*⟩ ≡ 776
```
int sampOffset = s*nLightPaths + i;
⟨Choose light source to trace virtual light path from 777⟩
⟨Sample ray leaving light source for virtual light path 778⟩
Intersection isect;
while (scene->Intersect(ray, &isect) && !alpha.IsBlack()) {
    ⟨Create virtual light and sample new ray for path 778⟩
}
arena.FreeAll();
```

`Distribution1D::SampleDiscrete()` returns a light number to use based on the given sample value and the light CDF computed above.

⟨*Choose light source to trace virtual light path from*⟩ ≡ 777
```
float lightPdf;
int ln = lightDistribution->SampleDiscrete(lightNum[sampOffset],
                                           &lightPdf);
Light *light = scene->lights[ln];
```

Given a particular light, the `Light::Sample_L()` routine gives the ray leaving the light, the radiance it is carrying, and the probability density for sampling the ray. The product of this probability density and the probability density for starting a path from the chosen

ComputeLightSamplingCDF() 709
Distribution1D 648
Distribution1D::SampleDiscrete() 650
Intersection 186
Light 606
MemoryArena::FreeAll() 1017
Scene::Intersect() 23
Scene::lights 23
Spectrum::IsBlack() 265

light, `lightPdf`, gives the overall probability density for sampling the ray from all of the lights.

After the initial ray from the light, the process for incrementally sampling the vertices of the paths is generally similar to the approach used in the `PathIntegrator`. Here, the `alpha` variable tracks the current weight of the path, accounting for radiance emitted by the light source along the initial ray and the product of BSDF terms and sampling densities (i.e., the current value of α in Equation (15.9)).

⟨*Sample ray leaving light source for virtual light path*⟩ ≡ 777
```
RayDifferential ray;
float pdf;
LightSample ls(lightSampPos[2*sampOffset], lightSampPos[2*sampOffset+1],
               lightSampComp[sampOffset]);
Normal Nl;
Spectrum alpha = light->Sample_L(scene, ls, lightSampDir[2*sampOffset],
                                 lightSampDir[2*sampOffset+1],
                                 camera->shutterOpen, &ray, &Nl, &pdf);
if (pdf == 0.f || alpha.IsBlack()) continue;
alpha /= pdf * lightPdf;
```

At each path vertex, the BSDF is found and the ray's weight is updated to account for attenuation through participating media. (Additional radiance due to in-scattering isn't included in the implementation here.)

⟨*Create virtual light and sample new ray for path*⟩ ≡ 777
```
alpha *= renderer->Transmittance(scene, RayDifferential(ray), NULL,
                                 rng, arena);
Vector wo = -ray.d;
BSDF *bsdf = isect.GetBSDF(ray, arena);
⟨Create virtual light at ray intersection point 779⟩
⟨Sample new ray direction and update weight for virtual light path 779⟩
```

Given the intersection at a surface in the scene, a `VirtualLight` object is created to represent the corresponding virtual light source. Given that `alpha` holds the path's contribution up to its arrival at the current point on the surface, we just need to store enough information so that the geometry term and two BSDF terms in that equation can be evaluated given a one-segment path from the camera. Thus, the point p_2 and surface normal at that point must be stored in addition to the `alpha` value, so that the geometric term $G(p_2 \leftrightarrow p_1)$ can be computed.

We'll make a further simplification to avoid needing to be able to evaluate the BSDF at the last vertex of the light path, $f(p_3 \rightarrow p_2 \rightarrow p_1)$. (In a sense, the product of this BSDF and the path's `alpha` vaue at p_2 can be thought of as defining the directionally varying radiant intensity of the virtual point light.) Here, we will represent this term with a constant Lambertian BSDF based on the hemispherical–directional reflectance of the surface, which makes it possible to include the $f(p_3 \rightarrow p_2 \rightarrow p_1)$ BSDF term in

BSDF 478
Intersection::GetBSDF() 484
LightSample 710
Normal 65
PathIntegrator 766
RayDifferential 69
Renderer::Transmittance() 25
Spectrum 263
Vector 57

the partial path contribution stored with the virtual light.[5] As long as the surfaces in the scene aren't too specular, the approximation made here works well.

⟨*Create virtual light at ray intersection point*⟩ ≡ 778
```
Spectrum contrib = alpha * bsdf->rho(wo, rng) / M_PI;
virtualLights[s].push_back(VirtualLight(isect.dg.p, isect.dg.nn, contrib,
                                        isect.rayEpsilon));
```

⟨*IGIIntegrator Private Data*⟩ +≡
```
vector<vector<VirtualLight> > virtualLights;
```

⟨*IGIIntegrator Local Structures*⟩ ≡
```
struct VirtualLight {
    VirtualLight(const Point &pp, const Normal &nn, const Spectrum &c,
                 float reps)
        : p(pp), n(nn), pathContrib(c), rayEpsilon(reps) { }
    Point p;
    Normal n;
    Spectrum pathContrib;
    float rayEpsilon;
};
```

Sampling the outgoing direction leaving an intersection and computing its updated weight are generally similar to the path sampling computation in the `PathIntegrator`. If the path continues, `alpha` is updated to incorporate the next term of the product term in the definition of α after Equation (15.9).

⟨*Sample new ray direction and update weight for virtual light path*⟩ ≡ 778
```
Vector wi;
float pdf;
BSDFSample bsdfSample(rng);
Spectrum fr = bsdf->Sample_f(wo, &wi, bsdfSample, &pdf);
if (fr.IsBlack() || pdf == 0.f)
    break;
Spectrum contribScale = fr * AbsDot(wi, bsdf->dgShading.nn) / pdf;
⟨Possibly terminate virtual light path with Russian roulette 780⟩
alpha *= contribScale / rrProb;
ray = RayDifferential(isect.dg.p, wi, ray, isect.rayEpsilon);
```

One small difference here compared to other integrators in this chapter is how Russian roulette is used for path termination. In the others, the termination probability in the Russian roulette test is generally constant after a fixed number of bounces have occurred. Here, the termination probability is one minus the luminance of the product of the BSDF's value and the cosine term divided by the sampling PDF for the outgoing direction. Thus, if the surface has a high albedo—it reflects most of the incident illumination—the probability of continuing the path is high, and if the albedo is small

AbsDot() 61
BSDF::dgShading 479
BSDF::rho() 482
BSDF::Sample_f() 706
BSDFSample 705
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
IGIIntegrator 773
Intersection::dg 186
Intersection::rayEpsilon 186
MemoryArena 1015
M_PI 1002
Normal 65
Point 63
RayDifferential 69
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VirtualLight 779

5 The alternative would be to retain the BSDFs, which would require not freeing the space allocated by the `MemoryArena` during preprocessing here; the arena would instead likely need to be a member variable of the `IGIIntegrator`.

it is likely that the path will be terminated. This is an intuitively efficient property; if the path is carrying a large amount of light, it should be likely to continue.

An interesting implication of this approach is that the luminance of the path contribution value remains constant after each surface intersection. For example, consider a surface where the luminance of `contribScale` is 0.1. The path will be terminated with 90% probability. For the 10% where the path continues, `alpha` will be scaled by `contribScale`, to account for the light reflection, and divided by 0.1, to account for Russian roulette. Because the luminance of `contribScale` is 0.1, the net result will be that the luminance of the path is unchanged.

Thus, surfaces that reflect little light lead to fewer paths leaving them, rather than paths with a lower contribution, and all of the virtual lights created will have similar path contributions and will thus tend to make an equal contribution to the final exitant radiance values computed. This generally leads to better results than if some of them were thousands of times brighter than others, for example.

⟨*Possibly terminate virtual light path with Russian roulette*⟩ ≡ 779
```
float rrProb = min(1.f, contribScale.y());
if (rng.RandomFloat() > rrProb)
    break;
```

15.4.2 RENDERING WITH VIRTUAL LIGHT SOURCES

Most of the `IGIIntegrator::Li()` implementation isn't included here; it samples direct lighting in the usual manner, recursively tracing rays to account for perfect specular reflection and refraction, as integrators like `DirectLightingIntegrator` do. After the direct lighting contribution has been computed at the point being shaded, the contribution from the virtual light sources is found by the ⟨*Compute indirect illumination with virtual lights*⟩ fragment. This fragment starts by using the appropriate sample value from `sample` to determine which of the sets of virtual lights to use for this point.

⟨*Compute indirect illumination with virtual lights*⟩ ≡
```
uint32_t lSet = min(uint32_t(sample->oneD[vlSetOffset][0] * nLightSets),
                    nLightSets-1);
for (uint32_t i = 0; i < virtualLights[lSet].size(); ++i) {
    const VirtualLight &vl = virtualLights[lSet][i];
    ⟨Compute virtual light's tentative contribution Llight 782⟩
    ⟨Possibly skip virtual light shadow ray with Russian roulette 782⟩
    ⟨Add contribution from VirtualLight vl 782⟩
}
if (ray.depth < maxSpecularDepth) {
    ⟨Do bias compensation for bounding geometry term 783⟩
}
```

Since the BSDF of the last vertex of the light path was approximated by a constant Lambertian term, there are two remaining factors to evaluate from Equation (15.9) in order to compute a light path's contribution to the current camera path:

$$f(p_0 \to p_1 \to p_2) G(p_1 \leftrightarrow p_2).$$

RNG::RandomFloat() 1003
Spectrum::y() 273
VirtualLight 779

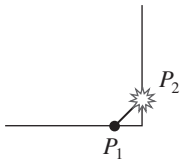
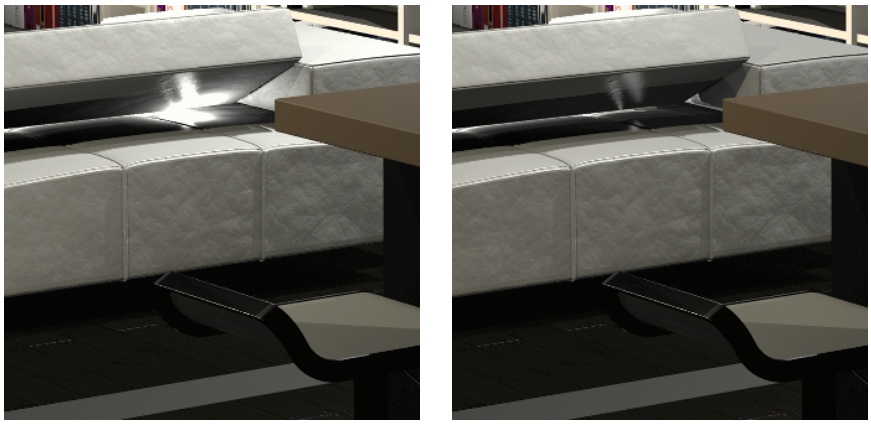


Figure 15.12: The Weak Singularity from the $G(p_1 \leftrightarrow p_2)$ Term. When the virtual light at p_2 and the point being shaded at p_1 are close together, this term can become very large due to the $1/r^2$ term, leading to bright splotches in rendered images. The `IGIIntegrator` clamps G to `gLimit` to avoid this problem, using a different sampling strategy that handles this case more robustly.



(a) (b)

Figure 15.13: Effect of the Weak Singularity When Evaluating Virtual Light Contributions. (a) If a virtual light is close to the point being shaded, the G term may be large, leading to bright splotches in the image. (b) Clamping G to be no larger than a fixed value eliminates these splotches but makes the rendering algorithm biased. Note, for example, that the top of the cushion in the upper right has become dimmer.

These are easily computed with the values available to the integrator and in the `VirtualLight` structure.

Here, we can see where the `gLimit` parameter is used by the implementation: if the points p_1 and p_2 are close to each other, as illustrated in Figure 15.12, G may become very large, leading to bright splotches in the image, as shown by Figure 15.13(a). For now, G is clamped so that it is no larger than `gLimit`, which eliminates these artifacts. However, this clamping makes the algorithm biased; images will be slightly darker on average than they should be (Figure 15.13(b)). The fragment ⟨*Do bias compensation for bounding geometry term*⟩, to be introduced shortly, accounts for this difference, making the final result again unbiased.

⟨*Compute virtual light's tentative contribution* `Llight`⟩ ≡ 780
```
float d2 = DistanceSquared(p, vl.p);
Vector wi = Normalize(vl.p - p);
float G = AbsDot(wi, n) * AbsDot(wi, vl.n) / d2;
G = min(G, gLimit);
Spectrum f = bsdf->f(wo, wi);
if (G == 0.f || f.IsBlack()) continue;
Spectrum Llight = f * G * vl.pathContrib / virtualLights[lSet].size();
RayDifferential connectRay(p, wi, ray, isect.rayEpsilon,
                           sqrtf(d2) * (1.f - vl.rayEpsilon));
Llight *= renderer->Transmittance(scene, connectRay, NULL, rng, arena);
```

If the light's contribution to outgoing radiance at the point is low, Russian roulette is used to sometimes avoid tracing the shadow ray for it. Thus, less time is spent tracing shadow rays for unimportant lights.

⟨*Possibly skip virtual light shadow ray with Russian roulette*⟩ ≡ 780
```
if (Llight.y() < rrThreshold) {
    float continueProbability = .1f;
    if (rng.RandomFloat() > continueProbability)
        continue;
    Llight /= continueProbability;
}
```

Finally, the visibility term is checked and the virtual light's contribution is added if it is not occluded.

⟨*Add contribution from* `VirtualLight vl`⟩ ≡ 780
```
if (!scene->IntersectP(connectRay))
    L += Llight;
```

To understand how to compensate for clamping the value of G above, consider the general setting of computing an area integral of some function f scaled by a geometric term,

$$\int_A f(\mathrm{p}, \mathrm{p}')\, G(\mathrm{p} \leftrightarrow \mathrm{p}')\, \mathrm{d}A. \tag{15.10}$$

When performing Monte Carlo integration, we'd like to clamp G when it becomes too large in order to avoid artifacts when a sample p' happens to be very close to p. To do so in a way that doesn't introduce bias, we can first use the identity $a = \min(a, b) + \max(a - b, 0)$ to rewrite Equation (15.10) as

$$\int_A f(\mathrm{p}, \mathrm{p}')\, \min(G(\mathrm{p}_\mathrm{i} \leftrightarrow \mathrm{p}_1), G_\mathrm{limit})\, \mathrm{d}A + \int_A f(\mathrm{p}, \mathrm{p}')\, \max(G(\mathrm{p}_\mathrm{i} \leftrightarrow \mathrm{p}_1) - G_\mathrm{limit}, 0)\, \mathrm{d}A. \tag{15.11}$$

The first term doesn't exhibit the weak singularity from G. We can then rewrite the second term as an integral over directions rather than area, using the Jacobian for the relationship between surface area and solid angle, $r^2/|\cos\theta|$,

AbsDot() 61
BSDF::f() 481
DistanceSquared() 65
IGIIntegrator::gLimit 774
IGIIntegrator::rrThreshold 774
IGIIntegrator::virtualLights 779
Intersection::rayEpsilon 186
RayDifferential 69
Renderer::Transmittance() 25
RNG::RandomFloat() 1003
Scene::IntersectP() 24
Spectrum 263
Spectrum::IsBlack() 265
Spectrum::y() 273
Vector 57
Vector::Normalize() 63
VirtualLight::pathContrib 779
VirtualLight::rayEpsilon 779

$$\int_{S^2} f(\mathrm{p}, \mathrm{p}') \max(G(\mathrm{p} \leftrightarrow \mathrm{p}') - G_{\mathrm{limit}}, 0) \frac{\| \mathrm{p} - \mathrm{p}' \|^2}{|\cos\theta|} \, \mathrm{d}\omega'$$

where $\mathrm{p}' = t(\mathrm{p}, \omega')$, using the trace function t from Section 15.2.1. This expression can be rewritten using the definition of G as

$$\int_{S^2} f(\mathrm{p}, \mathrm{p}') \frac{\max(G(\mathrm{p} \leftrightarrow \mathrm{p}') - G_{\mathrm{limit}}, 0)}{G(\mathrm{p} \leftrightarrow \mathrm{p}')} |\cos\theta'| \, \mathrm{d}\omega', \tag{15.12}$$

where θ' is the angle between the ray from p to p′ with the surface normal at p′. Note that this form doesn't exhibit the weak singularity as p′ approaches p.

Now we can apply the approach embodied by Equation (15.12) to the problem at hand. When we use the virtual light sources to evaluate Equation (15.9) to compute indirect illumination at a point, we're evaluating an estimate of the path contribution function, Equation (15.6). It just so happens that we're using the same sets of samples $\mathrm{p}_n, \ldots, \mathrm{p}_2$ for all points being shaded. By clamping G in this computation, we have evaluated the first term of Equation (15.11), with f defined appropriately. Thus, we now need to add in the effect of Equation (15.12); this task is handled by ⟨*Do bias compensation for bounding geometry term*⟩, using the BSDF's sampling method to choose directions.

⟨*Do bias compensation for bounding geometry term*⟩ ≡ 780
```
int nSamples = (ray.depth == 0) ? nGatherSamples : 1;
for (int i = 0; i < nSamples; ++i) {
    Vector wi;
    float pdf;
    BSDFSample bsdfSample = (ray.depth == 0) ?
        BSDFSample(sample, gatherSampleOffset, i) : BSDFSample(rng);
    Spectrum f = bsdf->Sample_f(wo, &wi, bsdfSample,
                                &pdf, BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
    if (!f.IsBlack() && pdf > 0.f) {
        ⟨Trace ray for bias compensation gather sample 783⟩
        ⟨Add bias compensation ray contribution to radiance sum 784⟩
    }
}
```

AbsDot() 61
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
IGIIntegrator::gatherSampleOffset 775
IGIIntegrator::nGatherSamples 774
Intersection 186
Intersection::rayEpsilon 186
Ray::depth 67
RayDifferential 69
Renderer::Li() 25
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

These rays are relatively inexpensive to trace since their maximum length can be bounded; given a particular value of `gLimit`, we can compute the distance beyond which G will always be less than `gLimit`, such that the value of the second term of Equation (15.11) will be zero. The ray's maximum distance is set to this value; thus, many of the rays often won't intersect any geometry, so no shading calculation needs to be performed for them.

⟨*Trace ray for bias compensation gather sample*⟩ ≡ 783
```
float maxDist = sqrtf(AbsDot(wi, n) / gLimit);
RayDifferential gatherRay(p, wi, ray, isect.rayEpsilon, maxDist);
Intersection gatherIsect;
Spectrum Li = renderer->Li(scene, gatherRay, sample, rng, arena,
                           &gatherIsect);
if (Li.IsBlack()) continue;
```

⟨*Add bias compensation ray contribution to radiance sum*⟩ ≡ 783

```
float Ggather = AbsDot(wi, n) * AbsDot(-wi, gatherIsect.dg.nn) /
    DistanceSquared(p, gatherIsect.dg.p);
if (Ggather - gLimit > 0.f && !isinf(Ggather)) {
    float gs  = (Ggather - gLimit) / Ggather *
                AbsDot(-wi, gatherIsect.dg.nn);
    L += f * Li * (AbsDot(wi, n) * gs / (nSamples * pdf));
}
```

15.5 IRRADIANCE CACHING

With unbiased light transport algorithms like path tracing, some scenes can take a large number of rays (and corresponding compute time) to generate images without objectionable noise. One approach to this problem has been the development of biased approaches to solving the LTE. These approaches generally reuse previously computed results over multiple exitant radiance computations, even when the values used don't give the precise quantity that needs to be computed (for example, by reusing an illumination value from a nearby point under the assumption that illumination is slowly changing). Irradiance caching, described in this section, and photon mapping, described in the next, have been two successful biased methods for light transport.

By introducing bias, these methods produce images without the high-frequency noise artifacts that unbiased Monte Carlo techniques are prone to. They can often create good-looking images using relatively little additional computation compared to basic techniques like Whitted ray tracing. This efficiency comes at a price, however: one key characteristic of unbiased Monte Carlo techniques is that variance decreases in a predictable and well-characterized manner as more samples are taken. As such, if an image was computed with an unbiased technique and has no noise, we can be extremely confident that the image correctly represents the lighting in the scene. With a biased solution method, however, error estimates aren't well defined for the approaches that have been developed so far; if the image doesn't have visual artifacts, it still may have error. Even worse, given an image with artifacts, increasing the sampling rate with a biased technique doesn't necessarily eliminate artifacts in a predictable way.

The *irradiance caching* algorithm is based on the observation that, while direct lighting often changes rapidly from point to point (e.g., consider a hard shadow edge), indirect lighting is often much more slowly changing. Therefore, if we compute an accurate representation of indirect light at a sparse set of sample points in the scene and then interpolate nearby samples to compute an approximate representation of indirect light at any particular point being shaded, we can expect that the error introduced by not recomputing indirect lighting everywhere shouldn't be too bad, and we can achieve a substantial computational savings by not recomputing this information at every point. Figure 15.14, which shows the indirect lighting component of a view of the San Miguel scene, illustrates the smoothness of indirect lighting in this environment.

DistanceSquared() 65
IGIIntegrator::gLimit 774

There are two issues that must be addressed in the design of an algorithm such as this one:



Figure 15.14: Indirect Illumination Component of a View of the San Miguel Scene. Note its smoothly varying and slowly changing nature. *(Model courtesy of Guillermo M. Leal Llaguno.)*

1. How is the indirect lighting distribution represented and stored after being computed at a point?
2. When are new representations of indirect light computed, and how often are already existing ones interpolated?

In the irradiance caching algorithm implemented here, indirect lighting is computed at a subset of the points in the scene that are shaded (as opposed to a predetermined set of points) and stored in a spatial data structure. When exitant radiance at a point is being computed, the cache is first searched for one or more acceptable nearby samples, using a set of error metrics to determine if the already existing samples are acceptable. If not enough acceptable samples are found, a new one is computed and added to the data structure.

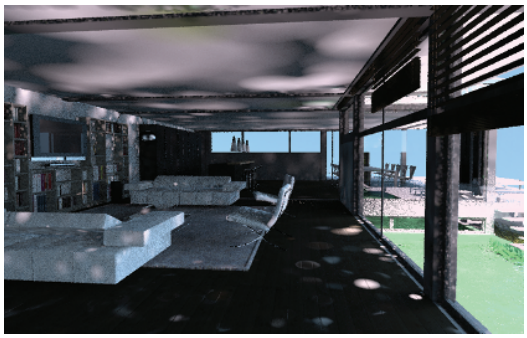
In order to have a compact representation of indirect light, this algorithm only stores the irradiance and the average direction of incident irradiance at each point, rather than a directionally varying radiance distribution, thus reducing the representation of incident illumination to just a single `Spectrum` and a single `Vector`.

To understand the main idea of the approach, first recall that irradiance arriving at one side of a surface with normal $\mathbf{n}$ is

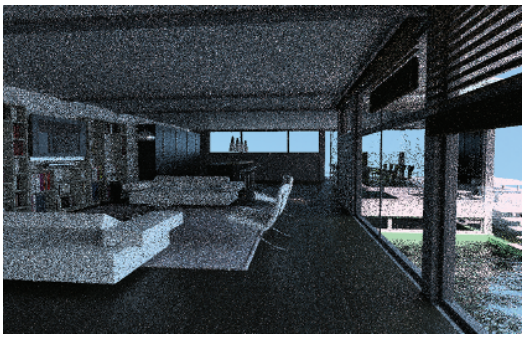
$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i}) \, |\cos \theta_\mathrm{i}| \, \mathrm{d}\omega_\mathrm{i}.$$

It is in a sense a weighted average of incoming radiance at a point, giving a sense of the aggregate illumination. Now consider the reflection component of the scattering equation for a reflective surface:

$$L_\mathrm{o}(\mathrm{p}, \omega_\mathrm{o}) = \int_{\mathcal{H}^2(\mathbf{n})} f_\mathrm{r}(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i}) \, L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i}) \, |\cos \theta_\mathrm{i}| \, \mathrm{d}\omega_\mathrm{i}.$$



(a)



(b)

Figure 15.15: Contemporary house interior scene rendered with (a) irradiance caching and (b) path tracing, both with approximately the same amount of computation time. We have intentionally limited the time spent rendering these in order to compare the image artifacts of the two methods when they are not able to take enough samples to sufficiently sample the scene radiance distribution. Path tracing has high-frequency noise, while irradiance caching suffers from circular blotches. *(Model courtesy of Florent Boyer.)*

If the surface is Lambertian, the BSDF is constant, and we have

$$\begin{aligned} L_o(\mathrm{p}, \omega_o) &= c \int_{\mathcal{H}^2(\mathbf{n})} L_i(\mathrm{p}, \omega_i) \, |\cos \theta_i| \, d\omega_i \\ &= c \, E(\mathrm{p}, \mathbf{n}). \end{aligned} \qquad [15.13]$$

In other words, for perfectly diffuse materials, the irradiance alone is enough information to exactly compute the reflection from the surface due to a particular incident lighting distribution. Thus, interpolating irradiance values can compute reflection from perfectly diffuse surfaces, with the only source of error being the error from interpolation. In the implementation below, the average incident lighting direction will be used to make the approach work well for mostly diffuse or slightly glossy surfaces; the details will be explained shortly.

Figure 15.15 shows the irradiance caching integrator described in this section in action. Figure 15.15(a) was rendered using the irradiance cache, and Figure 15.15(b) with path tracing, both using approximately the same amount of computation. The artifacts from undersampling with the irradiance cache are quite different than the variance from path tracing. Figure 15.16 shows the locations at which irradiance estimates were computed for this example.

⟨*IrradianceCacheIntegrator Declarations*⟩ ≡

```
class IrradianceCacheIntegrator : public SurfaceIntegrator {
public:
    ⟨IrradianceCacheIntegrator Public Methods⟩
private:
    ⟨IrradianceCacheIntegrator Private Data 787⟩
    ⟨IrradianceCacheIntegrator Private Methods⟩
};
```

IrradianceCacheIntegrator 786
SurfaceIntegrator 740

The `IrradianceCacheIntegrator` constructor initializes the member variables below with values passed into it; therefore, it is elided here. The sample pixel spacing member

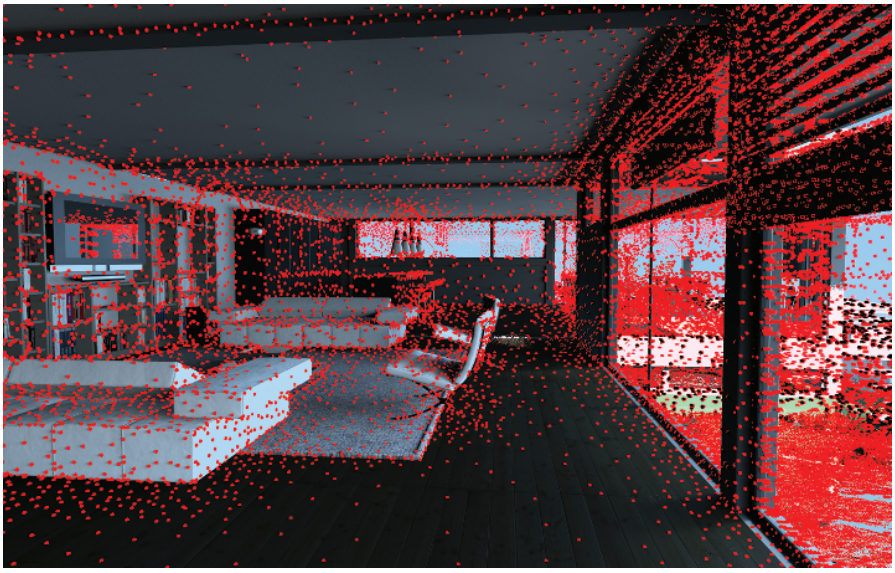


Figure 15.16: The Positions at Which Irradiance Estimates Were Computed for the Image in Figure 15.15. Note that they are mostly near the corners, where indirect illumination is changing most rapidly.

variables control the spacing of irradiance samples, measured in pixels on the image: they ensure that no two samples are closer than a minimum spacing as seen from the camera, but that no two are farther than a maximum. In general, extra samples less than a pixel or so apart have little visual impact on the image and aren't worth the computational expense, and not having enough samples at a minimum pixel frequency risks artifacts from excessive interpolation.

The minimum weight and `cosMaxSampleAngleDifference` values control the error metric used in determining if a particular irradiance sample can be used at a given point without introducing excessive error. Their use will be explained in the code that uses them. The number of Monte Carlo samples used to compute the irradiance estimates is controlled by `nSamples`, and the maximum depth parameters control the ray tree depths for specular reflections and the indirect lighting calculation done to compute irradiance samples. Finally, the class stores a reader–writer mutex to protect the octree storing irradiance samples from concurrent updates by multiple threads.

⟨*IrradianceCacheIntegrator Private Data*⟩ ≡ 786

```
float minSamplePixelSpacing, maxSamplePixelSpacing;
float minWeight, cosMaxSampleAngleDifference;
int nSamples, maxSpecularDepth, maxIndirectDepth;
mutable RWMutex *mutex;
```

`DirectLightingIntegrator` 742

`RWMutex` 1039

This integrator reuses the direct lighting routines that were defined in the direct lighting integrator, although here all lights are always sampled. Code fragments to request samples for this computation are reused from the `DirectLightingIntegrator` as well.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ ≡
```
void IrradianceCacheIntegrator::RequestSamples(Sampler *sampler,
        Sample *sample, const Scene *scene) {
    ⟨Allocate and request samples for sampling all lights 743⟩
}
```

⟨*IrradianceCacheIntegrator Private Data*⟩ +≡ 786
```
⟨Declare sample parameters for light source sampling 743⟩
```

The `Preprocess()` method allocates the octree that stores the irradiance samples. (The octree can't be allocated until the scene has been fully specified and its bounds are known—this isn't the case when the `IrradianceCacheIntegrator`'s constructor runs.) The method expands the bounds by a small amount in each direction so that the octree can gracefully deal with the fact that some of the irradiance samples and some of the lookup points may be marginally outside the scene bounds due to floating-point error from ray intersection computations.

After the octree is allocated, the ⟨*Prime irradiance cache*⟩ fragment executes. This fragment's job is to prefill the irradiance cache with irradiance samples that are likely to be needed when actually rendering the image. If the cache wasn't prefilled but was purely filled on demand, then we'd face the problem that often after a new irradiance sample was computed and added to the cache we would have liked to have had that sample available when computing outgoing radiance from previous points: two immediately adjacent image samples that should actually have the same incident irradiance used in shading them may instead use very different values—the first sample interpolating among existing irradiance values, just passing the error threshold, and the second not passing the error threshold for interpolation and thus computing an entirely new sample.

This fragment creates a `Sampler` that takes a single sample per pixel, decomposes the image plane into tiles, and launches a task for each tile. The tasks use the sampler to generate camera rays, find their intersections with objects in the scene, and call the `IrradianceCacheIntegrator::Li()` method. This fragment isn't included here because it is relatively long and similar in form to the `SamplerRenderer` and `SamplerRendererTasks` that it creates; the main difference is that the value returned from `Li()` is discarded; the method is called purely to cause cache samples to be added.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡
```
void IrradianceCacheIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    BBox wb = scene->WorldBound();
    Vector delta = .01f * (wb.pMax - wb.pMin);
    wb.pMin -= delta;
    wb.pMax += delta;
    octree = new Octree<IrradianceSample *>(wb);
    ⟨Prime irradiance cache⟩
}
```

⟨*IrradianceCacheIntegrator Private Data*⟩ +≡ 786
```
mutable Octree<IrradianceSample *> *octree;
```

BBox 70
BBox::pMax 71
BBox::pMin 71
Camera 302
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::Li() 789
IrradianceCacheIntegrator::octree 788
IrradianceSample 793
Octree 1023
Renderer 24
Sample 343
Sampler 340
SamplerRenderer 25
SamplerRendererTask 29
Scene 22
Scene::WorldBound() 24
Vector 57

15.5.1 RENDERING WITH THE IRRADIANCE CACHE

We won't include most of the irradiance cache's `Li()` method here—it does the usual BSDF evaluation and so forth—but will just focus on its key two fragments, ⟨*Compute direct lighting for irradiance cache*⟩ and ⟨*Compute indirect lighting for irradiance cache*⟩. For direct lighting, the `UniformSampleAllLights()` function from Section 15.1 is used to apply multiple importance sampling for the direct lighting estimate.

⟨*Compute direct lighting for irradiance cache*⟩ ≡
```
L += UniformSampleAllLights(scene, renderer, arena, p, n, wo,
         isect.rayEpsilon, ray.time, bsdf, sample, rng,
         lightSampleOffsets, bsdfSampleOffsets);
```

The `IrradianceCacheIntegrator` partitions the BSDF for the indirect lighting computation. Perfect specular reflection is handled by sampling the BSDF and recursively calling the integrator, just as the `WhittedIntegrator` does. The implementation here uses irradiance caching for both the diffuse and glossy components of the BSDF, thus introducing additional error for the glossy components. As long as the glossiness isn't too close to perfect specular reflection, the error introduced from this approximation is generally acceptable.

⟨*Compute indirect lighting for irradiance cache*⟩ ≡
```
if (ray.depth + 1 < maxSpecularDepth) {
    Vector wi;
    ⟨Trace rays for specular reflection and refraction 46⟩
}
⟨Estimate indirect lighting with irradiance cache 789⟩
```

If the surface has both reflective and transmissive nonspecular components, then reflection and transmission must be handled separately with two independent irradiance values, since the irradiance values from the hemisphere that is needed for reflective surfaces and the hemisphere on the opposite side of the surface that is needed for transmissive surfaces can be completely different. The `indirectLo()` method handles either case, interpolating a value using the cache or computing a new value and using it to compute exitant radiance due to incident irradiance.

Before calling this method, the integrator reorients the normal so that it points in the same hemisphere as the ω_o vector. `indirectLo()` depends on this convention when it generates sample rays over the hemisphere, since it will ensure that all of these rays are in the same hemisphere as `n`.

⟨*Estimate indirect lighting with irradiance cache*⟩ ≡ 789
```
Normal ng = isect.dg.nn;
ng = Faceforward(ng, wo);
⟨Compute pixel spacing in world space at intersection point 790⟩
BxDFType flags = BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE | BSDF_GLOSSY);
L += indirectLo(p, ng, pixelSpacing, wo, isect.rayEpsilon,
                bsdf, flags, rng, scene, renderer, arena);
flags = BxDFType(BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
L += indirectLo(p, -ng, pixelSpacing, wo, isect.rayEpsilon,
                bsdf, flags, rng, scene, renderer, arena);
```

BSDF_DIFFUSE 428
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDFType 428
DifferentialGeometry::nn 102
Faceforward() 66
Intersection::dg 186
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::
indirectLo() 790
IrradianceCacheIntegrator::
maxSpecularDepth 787
Normal 65
Ray::depth 67
UniformSampleAllLights() 745
Vector 57

Part of the error metric used in deciding which irradiance samples can be acceptably interpolated is based on the distance between a candidate sample and the current lookup point, expressed in pixels. Recall from Section 2.2.3 that the cross product of two vectors gives the area of the parallelogram that they form; taking the cross product of the $\partial p/\partial x$ and $\partial p/\partial y$ vectors from the `DifferentialGeometry` structure gives the world-space area of a pixel at the point being shaded. If we make the approximation that this region is square, then its square root gives the length of a side—the distance that we need to compute.

⟨*Compute pixel spacing in world space at intersection point*⟩ ≡ 789
```
float pixelSpacing = sqrtf(Cross(isect.dg.dpdx, isect.dg.dpdy).Length());
```

The `indirectLo()` method returns the reflected radiance for the BSDF components specified by the `flags` parameter, computed by finding the irradiance at the given point in the hemisphere by the given normal. A surprising number of additional parameters to the method are needed; many of them are needed for cases where a new irradiance value needs to be computed and added to the cache.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡
```
Spectrum IrradianceCacheIntegrator::indirectLo(const Point &p,
        const Normal &ng, float pixelSpacing, const Vector &wo,
        float rayEpsilon, BSDF *bsdf, BxDFType flags, RNG &rng,
        const Scene *scene, const Renderer *renderer,
        MemoryArena &arena) const {
    if (bsdf->NumComponents(flags) == 0)
        return Spectrum(0.);
    Spectrum E;
    Vector wi;
    ⟨Get irradiance E and average incident direction wi at point p 791⟩
    ⟨Compute reflected radiance due to irradiance and BSDF 791⟩
}
```

The ⟨*Get irradiance* `E` *and average incident direction* `wi` *at point* `p`⟩ fragment will be defined shortly, in the following subsection. Before going into the details of irradiance sample computation, caching, and interpolation, we'll first show how the computed irradiance estimates are used for shading.

Following the approach introduced by Tabellion and Lamorlette (2004), we can approximate the incident irradiance by incident radiance along a single direction—in some sense converting it to an equivalent directional light source. Doing so maintains a very rough approximation of the directional distribution of incident radiance at the point being shaded. The end result for perfectly diffuse surfaces is the same as if we just multiplied the irradiance by the reflectance, but for moderately glossy surfaces this approximation produces better results than assuming a uniform incident distribution of illumination.

Given an irradiance value E and an incident direction ω_{avg}, we can compute the amount of radiance L that must arrive along that direction to give the same irradiance:

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L(\mathrm{p}, \omega_{\mathrm{i}})\, \delta(\omega_{\mathrm{i}} - \omega_{\text{avg}})\, |\cos\theta_i|\, \mathrm{d}\omega_{\mathrm{i}} = L_{\text{avg}}\, |\cos\theta_{\text{avg}}|,$$

BSDF 478
BSDF::NumComponents() 479
BxDFType 428
Cross() 62
DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
Intersection::dg 186
IrradianceCacheIntegrator 786
MemoryArena 1015
Normal 65
Point 63
Renderer 24
RNG 1003
Scene 22
Spectrum 263
Vector 57
Vector::Length() 62

so $L_{\text{avg}} = E/|\cos\theta_{\text{avg}}|$. Substituting into the scattering equation, we have

$$\begin{aligned} L_{\text{o}}(\text{p}, \omega_{\text{o}}) &= \int_{\mathcal{H}^2(\mathbf{n})} f(\text{p}, \omega_{\text{o}}, \omega_{\text{i}})\, \delta(\omega_{\text{i}} - \omega_{\text{avg}})\, \frac{E}{|\cos\theta_{\text{avg}}|}\, |\cos\theta_{\text{i}}|\, \text{d}\omega_{\text{i}} \\ &= f(\text{p}, \omega_{\text{o}}, \omega_{\text{avg}}) E(\text{p}, \mathbf{n}). \end{aligned}$$

Thus, computing the reflected radiance is straightforward. The only complication is that the length of the weighted average direction of incident radiance function may be zero; this happens when there is no irradiance at the point. In this case, a black spectrum is returned immediately to avoid calling `Vector::Normalize()` on a zero-length vector, which would in turn return not-a-number values.

⟨*Compute reflected radiance due to irradiance and BSDF*⟩ ≡ 790
```
if (wi.LengthSquared() == 0.f) return Spectrum(0.);
return bsdf->f(wo, Normalize(wi), flags) * E;
```

15.5.2 LOOKUP AND INTERPOLATITON

The `interpolateE()` method attempts to interpolate the irradiance values already in the octree to approximate the irradiance at the given point in the hemisphere with the given normal. If suitable irradiance samples aren't found in the cache, a new sample is computed and added. In either case, `E` is initialized with an approximation of irradiance for this point, `wi` is initialized to the average direction of incident radiance, and exitant radiance can be computed with the approximation developed above. The fragments that define the computation to compute new irradiance samples are defined in Section 15.5.3; this section will focus on the lookup and interpolation process.

⟨*Get irradiance* `E` *and average incident direction* `wi` *at point* `p`⟩ ≡ 790
```
if (!interpolateE(scene, p, ng, &E, &wi)) {
    ⟨Compute irradiance at current point 795⟩
    ⟨Add computed irradiance value to cache 796⟩
}
```

Recall from the `Preprocess()` method that `IrradianceCacheIntegrator` uses an octree data structure to store the estimates. Doing so allows it to efficiently search for all of the already computed irradiance estimates around a point in the scene. The `Octree` template class, which is described in Section A.7 in Appendix A, recursively splits a given bounding box into subregions, refining the current region into eight subregions at each level of the tree by dividing the box in half at the midpoint of its extent along the x, y, and z axes. Each irradiance estimate has an axis-aligned bounding box associated with it, giving the overall area for which it is potentially a valid sample. The octree uses the extent of this box as a guide for an appropriate level of the tree at which to store the sample. Later, given a point to look up nearby irradiance samples for, the octree just needs to traverse the nodes that the point is inside (there is one such node at each level of the tree) and provide the samples overlapping those nodes to be considered for interpolation (Figure 15.17).

Much of the work in the `interpolateE()` method is done by the `Octree::Lookup()` method, which traverses the nodes of the octree that the given point is inside and calls the `operator()` method of the `IrradProcess` object for each `IrradianceSample` in each

BSDF::f() 481
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::interpolateE() 792
Spectrum 263
Vector::LengthSquared() 62
Vector::Normalize() 63

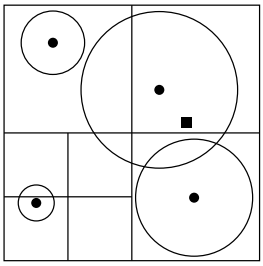


Figure 15.17: Example of Irradiance Sample Storage in 2D (with a Quadtree Rather than an Octree). Each irradiance sample, denoted by a dot, has a maximum distance over which it potentially can contribute irradiance, denoted here by a circle. Samples are stored in the tree nodes that they overlap, and the tree is refined adaptively so that each sample is stored in a small number of nodes. Given a point at which we want to look up nearby irradiance estimates, here shown with a black square, we just need to traverse the tree nodes that the point overlaps, considering all of the irradiance samples stored in these nodes.

of these nodes. `IrradProcess` decides if each sample is acceptable and accumulates the value of the interpolated result. Note that a reader lock is acquired before the lookup is performed; doing so ensures that another thread can't insert new values into the octree as it is being traversed in this thread.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡

```
bool IrradianceCacheIntegrator::interpolateE(const Scene *scene,
        const Point &p, const Normal &n, Spectrum *E,
        Vector *wi) const {
    if (!octree) return false;
    IrradProcess proc(p, n, minWeight, cosMaxSampleAngleDifference);
    RWMutexLock lock(*mutex, READ);
    octree->Lookup(p, proc);
    if (!proc.Successful()) return false;
    *E = proc.GetIrradiance();
    *wi = proc.GetAverageDirection();
    return true;
}
```

The `IrradProcess` structure stores the position and normal of the point being shaded as well as the values of the error parameters. It accumulates information about the interpolated value as it processes candidate irradiance samples. Its constructor is straightforward and not included here; it initializes the member variables with the parameter values given or sets them to initial values.

⟨*IrradianceCacheIntegrator Local Declarations*⟩ ≡

```
struct IrradProcess {
    ⟨IrradProcess Public Methods 794⟩
    ⟨IrradProcess Data 793⟩
};
```

IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::
 octree 788
IrradProcess 792
IrradProcess::
 GetAverageDirection() 794
IrradProcess::
 GetIrradiance() 794
IrradProcess::
 Successful() 794
Normal 65
Octree::Lookup() 1027
Point 63
RWMutexLock 1039
Scene 22
Spectrum 263
Vector 57

⟨*IrradProcess Data*⟩ ≡ 792
```
Point p;
Normal n;
float minWeight, cosMaxSampleAngleDifference, sumWt;
int nFound;
Spectrum E;
Vector wAvg;
```

The `IrradianceSample` class stores all of the relevant information to represent an irradiance sample—the irradiance value, the position and normal where it was computed, and the average direction of incident illumination. Finally, its `maxDist` member variable stores the distance beyond which the sample absolutely should not be used for shading; its value is set based on information collected when the irradiance sample is computed below.

⟨*IrradianceCacheIntegrator Local Declarations*⟩ +≡
```
struct IrradianceSample {
    ⟨IrradianceSample Constructor⟩
    Spectrum E;
    Normal n;
    Point p;
    Vector wAvg;
    float maxDist;
};
```

Each time the `IrradProcess` callback method is called by `Octree::Lookup()`, it is passed a pointer to an irradiance sample from the octree. The method computes an error term that tries to estimate the error from using the sample at the shading point, which is compared to user-supplied error limits; if the sample passes the test, it's added to a running weighted sum of samples that contribute to the point being shaded.

⟨*IrradianceCacheIntegrator Method Definitions*⟩ +≡
```
bool IrradProcess::operator()(const IrradianceSample *sample) {
    ⟨Compute estimate error term and possibly use sample 794⟩
    return true;
}
```

Two error terms are computed to determine whether a particular candidate irradiance sample can reasonably be used at the lookup point. The first, `perr`, is based on the ratio of the distance from the lookup point to the point where the sample was computed to the sample's maximum allowed distance value. The maximum distance is set using a few criteria; see the fragment ⟨*Compute irradiance sample's contribution extent and bounding box*⟩ in a few pages for details.

The second error term, `nerr`, is based on the angle between the surface normal of the lookup point and the normal direction used for computing the irradiance sample. The estimated error increases as the angle approaches the maximum allowed angle. For efficiency, an error term based on the cosine of the angle is used, so that only a dot product is needed to compare their orientations.

IrradianceSample 793
IrradProcess 792
Normal 65
Point 63
Spectrum 263
Vector 57

Samples with acceptable error are then added to the weighted averages. Each sample is weighted by one minus its error, thus ensuring that low-error samples have higher contribution to the final interpolated result.

⟨*Compute estimate error term and possibly use sample*⟩ ≡ 793
```
float perr = Distance(p, sample->p) / sample->maxDist;
float nerr = sqrtf((1.f - Dot(n, sample->n)) /
                   (1.f - cosMaxSampleAngleDifference));
float err = max(perr, nerr);
if (err < 1.) {
    float wt = 1.f - err;
    E += wt * sample->E;
    wAvg += wt * sample->wAvg;
    sumWt += wt;
}
```

When octree traversal and candidate sample processing is finished, it is necessary to decide if an acceptable interpolated irradiance value has been computed from the irradiance samples. If the sum of weights isn't greater than a user-supplied minimum weight, then we won't use the interpolated value; doing so improves the overall results by preventing the use of a small number of samples with high error.

⟨*IrradProcess Public Methods*⟩ ≡ 792
```
bool Successful() {
    return sumWt >= minWeight;
}
```

The final interpolated irradiance value is a weighted sum of the irradiance values of the acceptable estimates,

$$E = \frac{\sum_i w_i E_i}{\sum_i w_i}.$$

The average direction is just returned directly; the calling code will normalize it as needed.

⟨*IrradProcess Public Methods*⟩ +≡ 792
```
Spectrum GetIrradiance() const { return E / sumWt; }
Vector GetAverageDirection() const { return wAvg; }
```

15.5.3 ADDING NEW VALUES

If the `interpolateE()` method isn't able to find enough nearby irradiance samples of good enough quality, it computes a new irradiance value and adds it to the cache using the ⟨*Compute irradiance at current point*⟩ and ⟨*Add computed irradiance value to cache*⟩ fragments.

To compute a new irradiance estimate, we need to estimate the value of the integral

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i}) \, |\cos \theta_\mathrm{i}| \, \mathrm{d}\omega_\mathrm{i}. \quad [15.14]$$

IrradianceSample::E 793
IrradianceSample::maxDist 793
IrradianceSample::n 793
IrradianceSample::p 793
IrradianceSample::wAvg 793
IrradProcess::E 793
IrradProcess::minWeight 793
IrradProcess::n 793
IrradProcess::sumWt 793
IrradProcess::wAvg 793
Spectrum 263
Vector 57

Because there is no easy available way here to importance sample based on the distribution of incident radiance, the implementation uses a cosine-weighted distribution of directions. It is then faced with the problem of computing the amount of incident radiance along each one, $L_i(p, \omega_i)$—precisely the problem that all of the other integrators in this chapter address.

In the implementation here, the `IrradianceCacheIntegrator` uses standard path tracing to compute these values.[6] Before it starts tracing rays to compute the irradiance estimate, the integrator initializes a pair of random values to use to scramble the two dimensions of a low-discrepancy point sequence that it will map to cosine-weighted directions over the hemisphere. (See Section 7.4.3 for an explanation of how to randomly scramble point sequences so that a different set of sample values is used each time while still preserving the good distribution properties of the point set.) This method also tracks the minimum distance to the first intersection for all of the rays traced; this value is used in determining how widely the sample can be reused without excessive error.

To compute the irradiance estimate from the radiance values along the sample rays, the implementation uses the standard Monte Carlo estimator:

$$E(p, \mathbf{n}) = \frac{1}{N} \sum_j \frac{L_i(p, \omega_j)|\cos\theta_j|}{p(\omega_j)}.$$

Because the rays are sampled from a cosine-weighted distribution, $p(\omega) = \cos\theta/\pi$, so

$$\frac{1}{N} \sum_j \frac{L_i(p, \omega_j)|\cos\theta_j|}{|\cos\theta_j|/\pi} = \frac{\pi}{N} \sum_j L_i(p, \omega_j).$$

⟨*Compute irradiance at current point*⟩ ≡ 791

```
uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
float minHitDistance = INFINITY;
Vector wAvg(0,0,0);
Spectrum LiSum = 0.f;
for (int i = 0; i < nSamples; ++i) {
    ⟨Sample direction for irradiance estimate ray 796⟩
    ⟨Trace ray to sample radiance for irradiance estimate 796⟩
}
E = (M_PI / float(nSamples)) * LiSum;
```

Choosing the direction for the ray just requires generating the 2D low-discrepancy sample and calling `CosineSampleHemisphere()`. It returns a direction in the canonical reflection coordinate system, with the normal direction mapped to the $+z$ axis. To get a world-space ray direction, the convenient `BSDF::LocalToWorld()` method can be used. This direction may then also need to be flipped so that it lies in the same hemisphere as the normal that was passed in.

BSDF::LocalToWorld() 480
CosineSampleHemisphere() 669
INFINITY 1002
IrradianceCacheIntegrator 786
IrradianceCacheIntegrator::nSamples 787
M_PI 1002
RNG::RandomUInt() 1003
Spectrum 263
Vector 57

6 Using bidirectional path tracing to compute these values (to capture, for example, indirect lighting due to a spotlight shining at a small area on a ceiling) would be a good improvement to the basic algorithm implemented here.

⟨*Sample direction for irradiance estimate ray*⟩ ≡ 795
```
float u[2];
Sample02(i, scramble, u);
Vector w = CosineSampleHemisphere(u[0], u[1]);
RayDifferential r(p, bsdf->LocalToWorld(w), rayEpsilon);
r.d = Faceforward(r.d, ng);
```

Once the ray has been generated, path tracing is performed. The method that implements this computation, `pathL()`, is not included here, since it is essentially a stand-alone implementation of the `PathIntegrator`'s `Li()` method from Section 15.3.4. The returned incident radiance estimate is then added to the radiance sum here. The weighted average direction for incident radiance is computed, scaling the current ray's direction by the luminance of the radiance it carries and computing the sum, and then the distance to the intersection point is used to update the variable that tracks the minimum distance of all of the intersection points for this irradiance estimate.

⟨*Trace ray to sample radiance for irradiance estimate*⟩ ≡ 795
```
Spectrum L = pathL(r, scene, renderer, rng, arena);
LiSum += L;
wAvg += r.d * L.y();
minHitDistance = min(minHitDistance, r.maxt);
```

Now that the irradiance estimate has been computed, we need to compute an upper bound of the region of the scene it may provide values for in order to insert the sample into the octree.

⟨*Add computed irradiance value to cache*⟩ ≡ 791
```
⟨Compute irradiance sample's contribution extent and bounding box 797⟩
⟨Allocate IrradianceSample, get write lock, add to octree 797⟩
wi = wAvg;
```

There are three criteria that determine the region in which an irradiance sample may be valid. The first two criteria are based on pixel spacing constraints. As described above, the pixel spacing criteria ensure that, no matter what, we don't deposit samples at a rate greater than `minSamplePixelSpacing` as measured on the image plane, and that, no matter what, we place a sample at the minimum rate of `maxSamplePixelSpacing`. Those parameters are converted to world-space distances here by multiplying by the provided value of the length of the distance between pixels in world space at the point `pixelSpacing`.

The third criterion is based on the minimum distance to the closest intersection found when computing the irradiance sample. The idea is that if there is a nearby occluding object, then the indirect irradiance is likely to be changing quickly around the sample. For example, near the corner where a wall meets the ceiling, irradiance will in general be changing more rapidly than it will be in the middle of the ceiling. This term captures this effect, reducing the region of applicability for samples that are near occluders.

BSDF::LocalToWorld() 480
CosineSampleHemisphere() 669
Faceforward() 66
PathIntegrator 766
Ray::d 67
Ray::maxt 67
RayDifferential 69
Sample02() 372
Spectrum 263
Spectrum::y() 273
Vector 57

⟨*Compute irradiance sample's contribution extent and bounding box*⟩ ≡ 796
```
float maxDist = maxSamplePixelSpacing * pixelSpacing;
float minDist = minSamplePixelSpacing * pixelSpacing;
float contribExtent = Clamp(minHitDistance / 2.f, minDist, maxDist);
BBox sampleExtent(p);
sampleExtent.Expand(contribExtent);
```

Given the `sampleExtent` bounding box that bounds the sample's valid region, we can allocate an irradiance sample object and add it to the octree. Note that we acquire a write lock to the octree before updating it, thus ensuring that other threads aren't concurrently traversing the tree as it is being modified here.

⟨*Allocate* `IrradianceSample`, *get write lock, add to octree*⟩ ≡ 796
```
IrradianceSample *sample = new IrradianceSample(E, p, ng, wAvg,
                                                contribExtent);
RWMutexLock lock(*mutex, WRITE);
octree->Add(sample, sampleExtent);
```

15.6 PARTICLE TRACING AND PHOTON MAPPING

Photon mapping is another approach for solving the LTE based on a biased algorithm. It handles both glossy and diffuse reflection well; perfectly specular reflection is handled in the usual manner with recursive ray tracing.

Photon mapping is one of a family of *particle-tracing* algorithms, which are based on the idea of constructing paths from the lights where, at each vertex of the path, the amount of incident illumination arriving at the vertex is recorded as an illumination sample.[7] After a certain number of these illumination samples have been computed, a data structure that stores a representation of the distribution of light in the scene is built; at rendering time, this representation is used to compute values of measurements needed to compute the image. Because the particle-tracing step is decoupled from computing the measurements, many measurements may be able to reuse the work done for a single particle path, thus leading to more efficient rendering algorithms.

In this section, we will start by introducing a theory of particle-tracing algorithms and will discuss the conditions that must be fulfilled by a particle-tracing algorithm so that arbitrary measurements can be computed correctly using the particles created by the algorithm. We will then describe an implementation of a photon mapping integrator that uses particles to estimate illumination by interpolating lighting contributions from particles around the point being shaded.

BBox 70
BBox::Expand() 72
Clamp() 1000
IrradianceCacheIntegrator::
maxSamplePixelSpacing 787
IrradianceCacheIntegrator::
minSamplePixelSpacing 787
IrradianceCacheIntegrator::
octree 788
IrradianceSample 793
Octree::Add() 1024
RWMutexLock 1039

7 The particle tracing formalism can also be used to understand the "instant global illumination" approach of Section 15.4. Introducing IGI as a variant of bidirectional path tracing, with light paths reused many times, was an equally accurate and somewhat gentler introduction.

★ 15.6.1 THEORETICAL BASIS FOR PARTICLE TRACING

Particle-tracing algorithms in computer graphics are often explained in terms of packets of energy being shot from the light sources in the scene that deposit energy at surfaces they intersect before scattering in new directions. This is an intuitive way of thinking about particle tracing, but the intuition that it provides doesn't make it easy to answer basic questions about how propagation and scattering affect the particles. For example, does their contribution fall off with squared distance like flux density? Or, which $\cos\theta$ terms, if any, affect particles after they scatter from a surface?

In order to give a solid theoretical basis for particle tracing, we will describe it using a framework introduced by Veach (1997, Appendix 4.A), which instead interprets the stored particle histories as samples from the scene's equilibrium radiance distribution. Under certain conditions on the distribution and weights of the particles, the particles can be used to compute estimates of nearly any measurement based on the light distribution in the scene. In this framework, it is quite easy to answer questions about the details of particle propagation like the ones earlier. After developing this theory here, the remainder of this section will demonstrate its application to photon mapping.

A particle-tracing algorithm generates a set of N samples of illumination at points p_j, on surfaces in the scene

$$(\mathrm{p}_j, \omega_j, \alpha_j),$$

where each sample records incident illumination from direction ω_j and has some weight α_j associated with it (Figure 15.18). We would like to determine the conditions on the weights and distribution of particle positions so that we can use them to correctly compute estimates of arbitrary measurements.

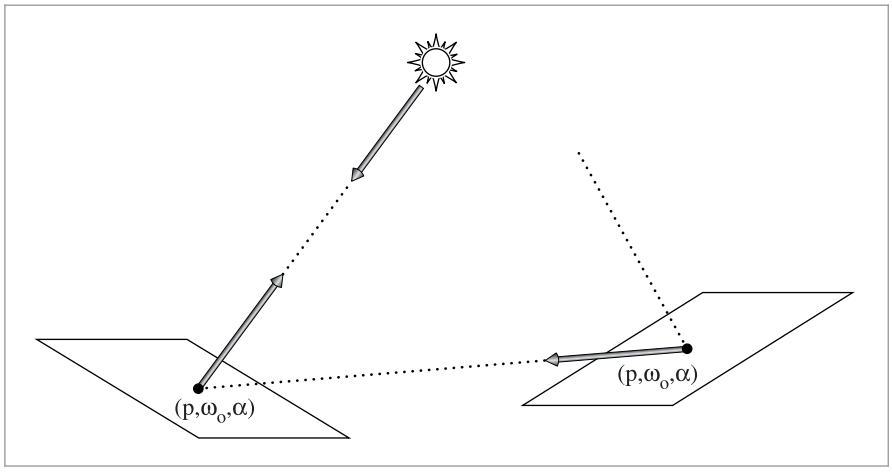


Figure 15.18: When a particle is traced following a path from a light source, an entry in its particle history is recorded at each surface it intersects. Each entry in the history is represented by position p, the direction of the ray it arrived along ω_o, and particle weight α.

Given an importance function $W_e(p, \omega)$ that describes the measurement to be taken, the natural condition we would like to be fulfilled is that the particles should be distributed and weighted such that using them to compute an estimate has the same expected value as the measurement equation for the same importance function:

$$E\left[\frac{1}{N}\sum_{j=1}^{N}\alpha_j W_e(p_j, \omega_j)\right] = \int_A \int_{S^2} W_e(p, \omega) L_i(p, \omega)\, dA\, d\omega. \quad [15.15]$$

For example, we might want to use the particles to compute the total flux on a wall. Using the definition of flux,

$$\Phi = \int_{A_{\text{wall}}} \int_{\mathcal{H}^2(\mathbf{n})} L_i(p, \omega)\, |\cos\theta|\, dA\, d\omega,$$

the following importance function selects the particles that lie on the wall and arrived from the hemisphere around the normal:

$$W_e(p, \omega) = \max((\omega \cdot \mathbf{n}), 0) \times \begin{cases} 1 & p \text{ is on wall surface} \\ 0 & \text{otherwise.} \end{cases}$$

If the conditions on the distribution of particle weights and positions are true for arbitrary importance functions such that Equation (15.15) holds, then the flux estimate can be computed directly as just a sum of the particle weights for the particles on the wall multiplied by the $(\omega \cdot \mathbf{n})$ term. If we want to estimate flux over a different wall, a subset of the original wall, and so on, we only need to recompute the weighted sum with an updated importance function. The particles and weights can be reused, and we have an unbiased estimate for all of these measurements. (The estimates will be correlated, however, which is potentially a source of artifacts.)

To see how to generate and weight particles that fulfill these conditions, consider the task of evaluating the measurement equation integral

$$\begin{aligned}&\int_A \int_{S^2} W_e(p_0, \omega)\, L(p_0, \omega)\, d\omega\, dA(p_0) \\ &\quad = \int_A \int_A W_e(p_0 \to p_1)\, L(p_1 \to p_0)\, G(p_0 \leftrightarrow p_1)\, dA(p_0)\, dA(p_1),\end{aligned}$$

where the importance function W_e that describes the measurement is a black box and thus cannot be used to drive the sampling of the integral at all. We can still compute an estimate of the integral with Monte Carlo integration, but must sample a set of points p_0 and p_1 from all of the surfaces in the scene, using some sampling distribution that doesn't depend on W_e (e.g., by uniformly sampling points by surface area).

By expanding the LTE in the integrand and applying the standard Monte Carlo estimator for N samples, we can find the estimator for this measurement,

$$E\left[\frac{1}{N}\sum_{i=1}^{N} W_e(p_{i,0} \to p_{i,1})\left\{\frac{L(p_{i,1} \to p_{i,0}) G(p_{i,0} \leftrightarrow p_{i,1})}{p(p_{i,0})\, p(p_{i,1})}\right\}\right].$$

We can further expand out the L term into the sum over paths and use the fact that $E[ab] = E[aE[b]]$ and the fact that for a particular sample, the expected value

$$E\left[\frac{L(\mathrm{p}_{i,1} \to \mathrm{p}_{i,0})}{p(\mathrm{p}_{i,0})}\right]$$

can be written as a finite sum of n_i terms in just the same way that we generated a finite set of weighted path vertices for path tracing. If the sum is truncated with Russian roulette such that the probability of continuing the sum after j terms is $q_{i,j}$, then the jth term of the ith sample has contribution

$$\frac{L_\mathrm{e}(\mathrm{p}_{n_i} \to \mathrm{p}_{n_i-1})}{p(\mathrm{p}_{n_i})} \prod_{j=1}^{n_i-1} \frac{1}{q_{i,j}} \frac{f(\mathrm{p}_{i,j+1} \to \mathrm{p}_{i,j} \to \mathrm{p}_{i,j-1}) G(\mathrm{p}_{i,j+1} \leftrightarrow \mathrm{p}_{i,j})}{p(\mathrm{p}_{i,j})}.$$

Looking back at Equation (15.15), we can see that this quantity gives the appropriate value of the particle weights if the particle-generating paths are sampled from a distribution over area:

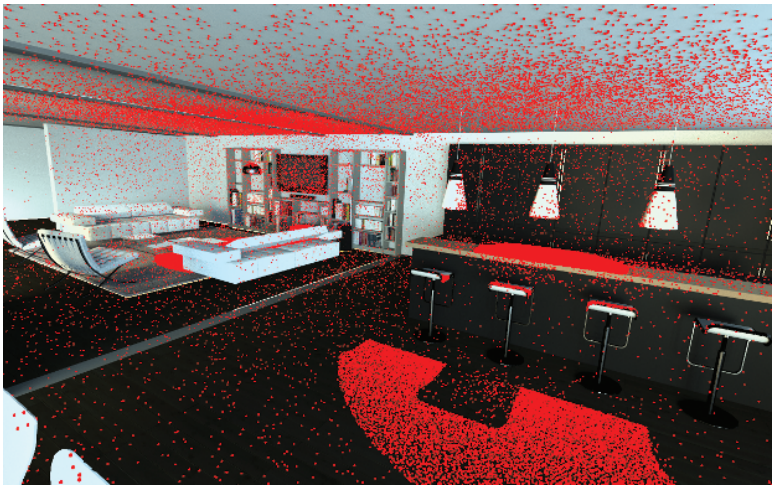
$$\alpha_{i,j} = \frac{L_\mathrm{e}(\mathrm{p}_{n_i} \to \mathrm{p}_{n_i-1})}{p(\mathrm{p}_{n_i})} \prod_{j=1}^{n_i-1} \frac{1}{q_{i,j}} \frac{f(\mathrm{p}_{i,j+1} \to \mathrm{p}_{i,j} \to \mathrm{p}_{i,j-1}) G(\mathrm{p}_{i,j+1} \leftrightarrow \mathrm{p}_{i,j})}{p(\mathrm{p}_{i,j})}. \quad \text{(15.16)}$$

Note that we have the freedom to generate a set of particles with these weights in all sorts of different ways. Although the natural approach is to start from points on lights and incrementally sample paths using the BSDFs at the path vertices, similar to how the path-tracing integrator generates paths (starting here from the light, rather than from the camera), we could generate them with any number of different sampling strategies, so long as there is nonzero probability of generating a particle at any point where W_e is nonzero and the particle weights are computed appropriately for the sampling distribution used.

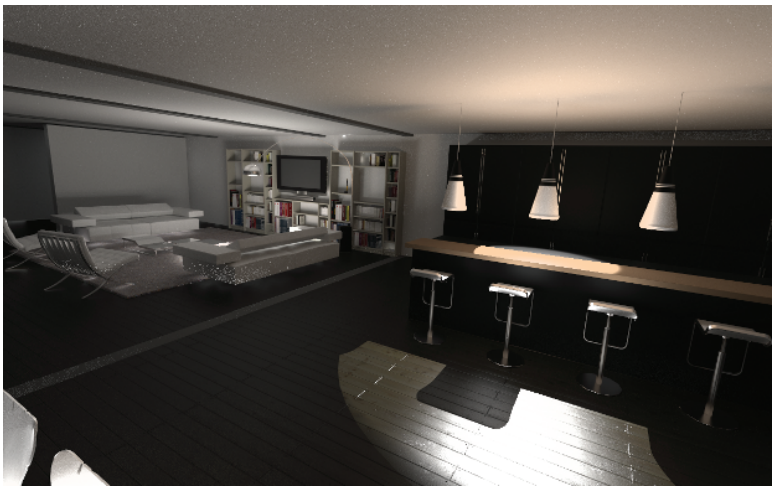
If we only had a single measurement to make, it would be better if we used information about W_e and could compute the estimate more intelligently, since the general particle-tracing approach described here may generate many useless samples if W_e only covers a small subset of the points on scene objects. If we will be computing many measurements, however, the key advantage that particle tracing brings is that we can generate the samples and weights once and can then reuse them over a large number of measurements, potentially computing results much more efficiently than if the measurements were all computed from scratch.

15.6.2 PHOTON INTEGRATOR

The photon mapping integrator traces particles into the scene and interpolates among particles to approximate the incident illumination at shading points. For consistency with other descriptions of the algorithm, we will refer to particles generated for photon mapping as photons. The integrator uses a kd-tree data structure to store the photons; this allows it to quickly find the photons around the point being shaded. The kd-tree is referred to as the *photon map*. Because the kd-tree is decoupled from the scene geometry, this algorithm isn't limited to a particular type of geometric representation (in contrast to using a texture map defined over shapes' (u, v) parameterizations to store illumina-



(a)



(b)

Figure 15.19: Rendering the Contemporary House Scene with Photon Mapping. (a) Locations of photons stored in the photon map. More photons are present in areas with brighter illumination. (b) Scene rendered using photon mapping. *(Model courtesy of Florent Boyer.)*

tion, for instance). Figure 15.19(a) shows the distribution of photons for a view of the contemporary house scene, and Figure 15.19(b) shows the scene rendered with photon mapping. Note that there is a greater density of photons in areas with more illumination.

Photon mapping partitions the LTE in a few ways that make it easier to adjust the quality of the results computed. For example, particles from the lights are characterized as being one of three types: direct illumination (light that has arrived directly at a surface, without any scattering), caustic illumination (light that has arrived at a nonspecular surface after

interacting with one or more specular surfaces), and indirect illumination (all other types of illumination). Thanks to this partitioning, the integrator can apply specific strategies to each type of illumination. In the implementation here, direct illumination is handled entirely with conventional direct lighting algorithms and caustics are always rendered directly with the photons in the photon map.

General indirect illumination is usually rendered using *final gathering*, where one bounce of path tracing is performed and the photon map is then used to compute reflection at the intersection points. (The photon map can also be used directly to approximate indirect illumination at shaded points without final gathering, though the visual results generally have artifacts.)

Quite a few different parameters control the operation of the integrator. The user must specify a desired number of photons of each type—caustic and indirect—to store in each of the two respective photon maps. Using more photons increases the quality of results but takes more time and memory. Because this integrator interpolates nearby photons to estimate illumination at the shading point, the user can also set how many photons are used for the interpolation. The more photons that are used, the smoother the illumination estimate will be, since a larger number of photons will be used to reconstruct it. If too many are used, the result will tend to be too blurry, while too few gives a splotchy appearance. Interpolating 50 to 100 photons is often a good choice. See the "Further Reading" section at the end of this chapter for references to research on more accurate methods of filtering photon contributions for reconstructing illumination.

We won't include the implementation of the `PhotonIntegrator` constructor here, since it just initializes its member variables from the parameters passed to it. Its member variables are the following:

- `nCausticPhotonsWanted` and `nIndirectPhotonsWanted` give the total number of photons to try to store for each category of illumination stored in the photon map. During preprocessing before rendering, ray paths will be followed until either the requested number of photons have been deposited or the system detects that it's not successfully depositing photons—for example, no caustic light-carrying paths will be found in a scene with no specular surfaces.
- `nLookup` gives the total number of photons to try to use for the interpolation step.
- `maxDistSquared` gives the maximum allowed squared distance from the point where exitant radiance is being computed to a photon that can be used for the interpolation there. If its value is too large, the integrator will waste time searching to find photons in dark regions; if it's too small, it may not be able to find `nLookup` nearby photons, leading to an overly splotchy result.
- `maxSpecularDepth` tracks the maximum path depth of specular reflection, similar to the Whitted integrator, and `maxPhotonDepth` can similarly limit the length of paths traced during the photon map construction step.
- `finalGather` controls if final gathering is used for indirect lighting rather than using the indirect photon map directly; if final gathering is enabled, `gatherSamples` controls the number of samples taken—64 to 128 samples generally work well if a single image sample per pixel is taken; as more pixel samples are taken, the number of gather samples for each one can be reduced correspondingly. If final gathering is enabled, then `cosGatherAngle` controls how the photons are used for importance

sampling directions for the final gather rays; they represent information about the directional distribution of incident illumination that can be used to improve the distribution of these rays.

⟨*PhotonIntegrator Private Data*⟩ ≡
```
uint32_t nCausticPhotonsWanted, nIndirectPhotonsWanted, nLookup;
float maxDistSquared;
int maxSpecularDepth, maxPhotonDepth;
bool finalGather;
int gatherSamples;
float cosGatherAngle;
```

The `PhotonIntegrator::RequestSamples()` method is similar to implementations in other integrators. It requests samples for multiple importance sampling for direct lighting and, if final gathering has been enabled, a well-distributed set of samples for sampling the BSDF for the final gather step.

⟨*PhotonIntegrator Method Definitions*⟩ ≡
```
void PhotonIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
        const Scene *scene) {
    ⟨Allocate and request samples for sampling all lights 743⟩
    ⟨Request samples for final gathering 803⟩
}
```

⟨*PhotonIntegrator Private Data*⟩ +≡
```
⟨Declare sample parameters for light source sampling 743⟩
```

In the implementation to follow, final gathering is done by sampling the BSDF to find directions for half of the final gather rays and sampling the indirect illumination distribution, as represented by the nearby photons' directions, for the other half. The `PhotonIntegrator::RequestSamples()` method therefore needs to request two sets of samples for final gathering; one will be used for the BSDF samples and the other for the samples taken using the nearby photons to define a sampling distribution.

⟨*PhotonIntegrator Private Data*⟩ +≡
```
BSDFSampleOffsets bsdfGatherSampleOffsets, indirGatherSampleOffsets;
```

⟨*Request samples for final gathering*⟩ ≡ 803
```
if (finalGather) {
    gatherSamples = max(1, gatherSamples/2);
    if (sampler) gatherSamples = sampler->RoundSize(gatherSamples);
    bsdfGatherSampleOffsets = BSDFSampleOffsets(gatherSamples, sample);
    indirGatherSampleOffsets = BSDFSampleOffsets(gatherSamples, sample);
}
```

BSDFSampleOffsets 706
PhotonIntegrator::finalGather 803
PhotonIntegrator::gatherSamples 803
Sample 343
Sampler 340
Sampler::RoundSize() 344
Scene 22

15.6.3 BUILDING THE PHOTON MAPS

When the `Preprocess()` method is called, particle paths are followed through the scene until the integrator has accumulated the desired number of particle histories to build the photon maps. Three separate photon maps are created: one to store photons that represent the direct illumination, one for indirect lighting, and one for caustics. (The

direct illumination photons are used only during this preprocessing step and are freed at the end of this method.) At each intersection of the path with an object, a weighted particle contribution is stored if the map for the corresponding type of illumination is not yet full. This phase of computation is executed in parallel; the concurrently running tasks must therefore carefully coordinate their updates of the shared data structures.

```
⟨PhotonIntegrator Method Definitions⟩ +≡
  void PhotonIntegrator::Preprocess(const Scene *scene,
          const Camera *camera, const Renderer *renderer) {
      if (scene->lights.size() == 0) return;
      ⟨Declare shared variables for photon shooting 804⟩
      ⟨Compute light power CDF for photon shooting 805⟩
      ⟨Run parallel tasks for photon shooting⟩
      ⟨Build kd-trees for indirect and caustic photons 813⟩
      ⟨Precompute radiance at a subset of the photons 815⟩
  }
```

A number of shared variables will be updated by the photon shooting tasks as they execute; references to them are passed into the tasks. Updates to these variables by the tasks must be protected with the mutex declared here. There are three arrays of `Photon` objects, which represent illumination samples (the contents of the `Photon` structure will be defined shortly). There is also an array of `RadiancePhotons`; they represent outgoing radiance at points in the scene, in contrast to the regular photons, which effectively represent incident illumination. The `RadiancePhotons` are used to accelerate final gathering computations; details of their use will be explained later in this section.

`abortTasks` is only set to `true` if one of the tasks determines that not enough photons are being stored with respect to the number of paths being traced; in this case, when the other tasks notice that it has been set, they exit.

Finally, the total number of paths started from the lights by all of the tasks is maintained in the `nshot` variable. In the end, this value may be larger or smaller than the sizes of the respective arrays of photons. On the one hand, it may be necessary to start more paths than photons are stored—for example, photons may leave the scene without intersecting any objects. On the other hand, each path may contribute multiple entries to the particle history as it bounces around the scene.

```
⟨Declare shared variables for photon shooting⟩ ≡ 804
  Mutex *mutex = Mutex::Create();
  int nDirectPaths = 0;
  vector<Photon> causticPhotons, directPhotons, indirectPhotons;
  vector<RadiancePhoton> radiancePhotons;
  bool abortTasks = false;
  uint32_t nshot = 0;
```

Camera 302
Mutex 1038
Photon 805
PhotonIntegrator 802
RadiancePhoton 815
Renderer 24
Scene 22
Scene::lights 23

The `Photon` structure stores just enough information to record a photon's contribution—the position where it hit the surface, its weight, and the direction it arrived from. Because hundreds of thousands or millions of photons may be stored, using a more compact

photon representation than the one used here can be worthwhile. To keep the implementation here simple, however, we will just use the straightforward representation below.

⟨*PhotonIntegrator Local Declarations*⟩ ≡
```
struct Photon {
    Photon(const Point &pp, const Spectrum &wt, const Vector &w)
        : p(pp), alpha(wt), wi(w) { }
    Point p;
    Spectrum alpha;
    Vector wi;
};
```

Recall that it is both the distribution and the weights of the collection of particles in a scene that together represent the equilibrium radiance distribution. For example, the light on a surface with a relatively large amount of illumination arriving at it could be represented by a small number of particles each with large weights, a large number of particles with smaller weights, or something between these two extremes. As far as the theory underlying the method, any of these representations is an equivalently reasonable representation of the scene's radiance distribution.

In practice, the photon mapping algorithm works best if all of the photons have the same (or similar) weights, and if it is only their varying density throughout the scene that represents the variation in illumination. If the photon weights have substantial variation, there can be unpleasant image artifacts. If one photon takes on a much larger weight than the others, it is easy to see the region of the scene where that photon contributes to the interpolated radiance: there is a bright circular artifact in that area.

Therefore, we'd like to shoot more photons from the brighter lights so that the initial weights of photons leaving all lights will be of similar magnitudes, and thus, the light to start each path from is chosen according to a PDF defined by the lights' respective power. Thus, it is a greater number of photons from the brighter lights that accounts for their greater contributions to illumination in the scene rather than a lower number of photons but with larger weights than the other lights. To implement this approach, a discrete PDF is computed to choose lights to shoot particles from based on each light's power.

⟨*Compute light power CDF for photon shooting*⟩ ≡ 804
```
Distribution1D *lightDistribution = ComputeLightSamplingCDF(scene);
```

The `PhotonShootingTask` handles the parallel creation of the photon maps. We won't include the code that launches the tasks, as it's essentially the same as other task launch code, and we've elided the declaration of the `PhotonShootingTask` class, which mostly just stores references to variables declared here and the parameters to the `Preprocess()` method so that they are available in the task's `Run()` method.

The `Run()` method stores photons in local arrays as it traces paths through the scene. After a few thousand paths have been followed, it acquires the mutex and updates the shared photon data structures. Compared to acquiring the mutex and updating the data structures each time a task deposits a photon, this approach is more efficient by amortizing the overhead of acquiring the mutex over more work.

ComputeLightSamplingCDF() 709
Distribution1D 648
Photon 805
Point 63
Spectrum 263
Vector 57

⟨*PhotonIntegrator Method Definitions*⟩ +≡
```
void PhotonShootingTask::Run() {
    ⟨Declare local variables for PhotonShootingTask 806⟩
    while (true) {
        ⟨Follow photon paths for a block of samples 807⟩
        ⟨Merge local photon data with data in PhotonIntegrator 812⟩
        ⟨Exit task if enough photons have been found 813⟩
    }
}
```

In addition to the local arrays for storing photons before they're merged to the shared arrays declared in `PhotonIntegrator::Preprocess()`, a `MemoryArena` and a `RNG` are needed. Note that the random number generator is seeded with a different value for each task so that each task follows different photon paths through the scene. `totalPaths` tracks the total number of photon paths that have been followed by this task, and `causticDone` and `indirectDone` are `false` until the requested number of photons of their respective types have been stored; they start out as `true` if zero of the corresponding type is needed.

⟨*Declare local variables for* `PhotonShootingTask`⟩ ≡ 806
```
MemoryArena arena;
RNG rng(31 * taskNum);
vector<Photon> localDirectPhotons, localIndirectPhotons, localCausticPhotons;
vector<RadiancePhoton> localRadiancePhotons;
uint32_t totalPaths = 0;
bool causticDone = (integrator->nCausticPhotonsWanted == 0);
bool indirectDone = (integrator->nIndirectPhotonsWanted == 0);
```

Six sample values are needed to start each path: one to select which light source to start the path from and five more for the `Light::Sample_L()` method that generates rays leaving the light. The fact that tasks are simultaneously generating samples with multiple threads of execution makes the sample generation problem more challenging than it would be otherwise. With a single-threaded implementation, we might use a Halton sequence for generating these samples, taking advantage of the fact that no matter when we stop generating more samples, in the aggregate, the samples would be well distributed. In a parallel implementation, we can't directly use the Halton sequence in each thread, since then all threads would generate the same set of photon paths! Therefore, the implementation here uses the randomized variant of the Halton sequence that was introduced in Section 7.4.2, using the `PermutedHalton` utility class, which is given the number of dimensions for which samples are needed as well as the uniquely seeded `RNG` for this task.

⟨*Declare local variables for* `PhotonShootingTask`⟩ +≡ 806
```
PermutedHalton halton(6, rng);
```

The task now traces `blockSize` paths from the lights, storing the resulting photons in the local arrays. Only after this phase has finished will the task acquire the mutex and merge its photons with the global data structures. To create a new path, the integrator samples a ray from one of the lights in the scene and then follows the path, recording

Light::Sample_L() 608
MemoryArena 1015
PermutedHalton 367
Photon 805
PhotonIntegrator::
nCausticPhotonsWanted 803
PhotonIntegrator::
nIndirectPhotonsWanted 803
RadiancePhoton 815
RNG 1003

particle intersections as it bounces around the scene. The `alpha` variable represents the particle weight from Equation (15.16) and is incrementally updated to store the path contribution at each vertex. After path termination, the BSDF memory allocated for the path and maintained by the `MemoryArena` is freed before going on to start the next one.

⟨*Follow photon paths for a block of samples*⟩ ≡ 806
```
const uint32_t blockSize = 4096;
for (uint32_t i = 0; i < blockSize; ++i) {
    float u[6];
    halton.Sample(++totalPaths, u);
    ⟨Choose light to shoot photon from 807⟩
    ⟨Generate photonRay from light source and initialize alpha 808⟩
    if (!alpha.IsBlack()) {
        ⟨Follow photon path through scene and record intersections 808⟩
    }
    arena.FreeAll();
}
```

Which light to start the path from is determined using the CDF based on light power computed previously. Here, the first component of the 6D sample from the randomly permuted Halton sequence, `u[0]`, is used.

⟨*Choose light to shoot photon from*⟩ ≡ 807
```
float lightPdf;
int lightNum = lightDistribution->SampleDiscrete(u[0], &lightPdf);
const Light *light = scene->lights[lightNum];
```

After the light has been chosen, its `Sample_L()` method is used to sample an outgoing ray. Given a light, a ray from the light source is sampled and its α value is initialized with

$$\alpha = \frac{|\cos \omega_0|\, L_e(\mathrm{p}_0, \omega_0)}{p(\mathrm{p}_0, \omega_0)},$$

where $p(\mathrm{p}_0, \omega_0)$ is the product of the probability for sampling this particular light and the product of the area and directional densities for sampling this particular ray leaving the light. The remaining five dimensions of the sample from the permuted Halton sequence are used for the sample values.

Note that it isn't possible to guarantee that all of the rays sampled by a light will have the same weights—some variation in photon weights may remain in spite of the efforts to choose lights according to their total power. For example, an area light with a texture map defining an illumination distribution might choose points uniformly over the light's surface rather than according to a PDF that matched the texture map's variation. In such a case, the weights for the rays from the lights would still be varying. Thus, though the implementation here does its best to create photons with equal weights, it's not possible to guarantee this property in all cases.

Distribution1D::SampleDiscrete() 650
Light 606
MemoryArena 1015
MemoryArena::FreeAll() 1017
PermutedHalton::Sample() 368
Spectrum::IsBlack() 265

⟨*Generate* photonRay *from light source and initialize* alpha⟩ ≡ 807
```
RayDifferential photonRay;
float pdf;
LightSample ls(u[1], u[2], u[3]);
Normal Nl;
Spectrum Le = light->Sample_L(scene, ls, u[4], u[5],
                              time, &photonRay, &Nl, &pdf);
if (pdf == 0.f || Le.IsBlack()) continue;
Spectrum alpha = (AbsDot(Nl, photonRay.d) * Le) / (pdf * lightPdf);
```

Now the integrator can start following the path through the scene, updating α after each scattering event and recording photons at the path vertices. The specularPath variable records whether the current path has only intersected perfectly specular surfaces after leaving the light source, which indicates that the path is a caustic path. Its value isn't used at the first intersection (direct lighting), so we can safely initialize its value to true here knowing that it will be set to false after any nonspecular bounce.

⟨*Follow photon path through scene and record intersections*⟩ ≡ 807
```
bool specularPath = true;
Intersection photonIsect;
int nIntersections = 0;
while (scene->Intersect(photonRay, &photonIsect)) {
    ++nIntersections;
    ⟨Handle photon/surface intersection 808⟩
    ⟨Sample new photon ray direction 810⟩
}
```

Given a photon–surface intersection, the particle weight must be updated to account for attenuation due to participating media from the last path vertex to this one. The photon may be stored in the appropriate array of photons, depending on whether more photons of the corresponding type are still required. However, if the current hit is a completely specular surface, there's no need to record it in any photon map, since we do not use photons for rendering reflections from specular surfaces.

⟨*Handle photon/surface intersection*⟩ ≡ 808
```
alpha *= renderer->Transmittance(scene, photonRay, NULL, rng, arena);
BSDF *photonBSDF = photonIsect.GetBSDF(photonRay, arena);
BxDFType specularType = BxDFType(BSDF_REFLECTION |
                        BSDF_TRANSMISSION | BSDF_SPECULAR);
bool hasNonSpecular = (photonBSDF->NumComponents() >
                       photonBSDF->NumComponents(specularType));
Vector wo = -photonRay.d;
if (hasNonSpecular) {
    ⟨Deposit photon at surface 810⟩
}
if (nIntersections >= integrator->maxPhotonDepth) break;
```

AbsDot() 61
BSDF 478
BSDF::NumComponents() 479
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDFType 428
Intersection 186
Intersection::GetBSDF() 484
Light::Sample_L() 608
LightSample 710
Normal 65
PhotonIntegrator::
maxPhotonDepth 803
Ray::d 67
RayDifferential 69
Renderer::Transmittance() 25
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

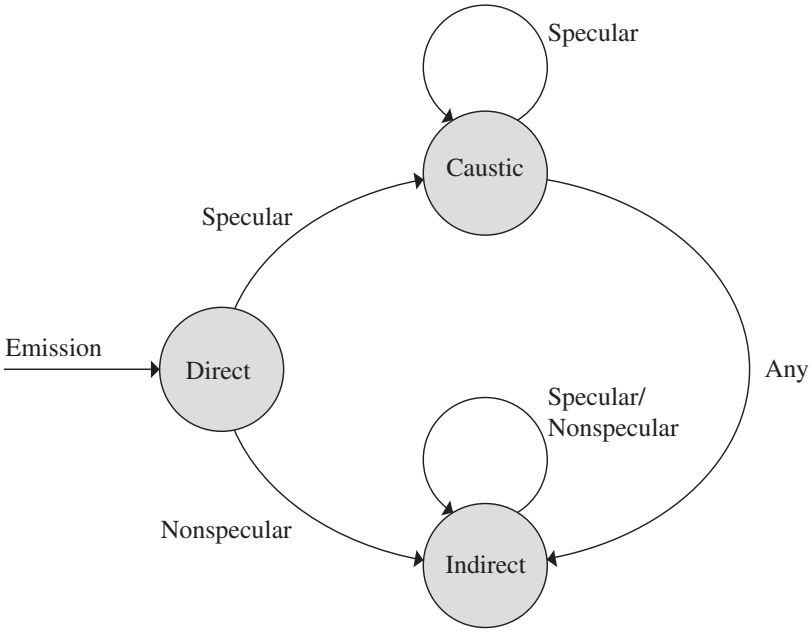


Figure 15.20: Finite-State Machine for Determining Which Photon Map to Update. An illustration of the transitions photons make from one state to another depending on the type of surface scattering they undergo at path vertices.

If this is the first intersection found after the particle has left the light source, then the photon represents direct illumination. Otherwise, if it has only reflected from specular surfaces before arriving at the current intersection point, it must be a caustic photon. Any other case—a path that only hit nonspecular surfaces or a path that hit a series of specular surfaces before scattering from nonspecular surfaces—represents indirect illumination. The finite-state machine diagram in Figure 15.20 illustrates the possibilities. The nodes of the graph show which of the photon maps should be updated for a photon in that state, and the edges describe whether the photon has scattered from a specular or a nonspecular BSDF component at its previous intersection. The code fragment here only includes the code for updating the caustic photon map; the code for direct and indirect photons is analogous and not included here.

As the photon maps are being populated, a random subset of photons is selected to also be `RadiancePhotons`, which store a precomputed outgoing radiance value. The fragment that creates them, ⟨*Possibly create radiance photon at photon intersection point*⟩, will be defined in a few pages, in the section that discusses radiance photons.

RadiancePhoton 815

⟨*Deposit photon at surface*⟩ ≡ 808
```
Photon photon(photonIsect.dg.p, alpha, wo);
bool depositedPhoton = false;
if (specularPath && nIntersections > 1) {
    if (!causticDone) {
        depositedPhoton = true;
        localCausticPhotons.push_back(photon);
    }
}
else {
    ⟨Deposit either direct or indirect photon⟩
}
⟨Possibly create radiance photon at photon intersection point 814⟩
```

Having recorded the particle's contribution in one of the photon maps (or ignoring it if that map type was full), the integrator needs to choose a new outgoing direction from the intersection point and update the α value to account for the effect of scattering of the incident illumination. Equation (15.15) shows how to incrementally update the particle weight after a scattering event: Given some weight $\alpha_{i,j}$ that represents the weight for the jth intersection of the ith particle history, after a scattering event where a new vertex $\mathrm{p}_{i,j+1}$ has been sampled, the weight should be set to be

$$\alpha_{i,j+1} = \alpha_{i,j} \frac{1}{q_{i,j+1}} \frac{f(\mathrm{p}_{i,j+1} \rightarrow \mathrm{p}_{i,j} \rightarrow \mathrm{p}_{i,j-1}) G(\mathrm{p}_{i,j+1} \leftrightarrow \mathrm{p}_{i,j})}{p(\mathrm{p}_{i,j+1})}.$$

As with the path-tracing integrator, there are a number of reasons to choose the next vertex in the path by sampling the BSDF's distribution at the intersection point to get a direction ω' and tracing a ray in that direction, rather than directly sampling by area on the scene surfaces. Therefore, we again apply the Jacobian to account for this change in measure, all of the terms in G except for a single $|\cos\theta|$ cancel out, and the expression is

$$\alpha_{i,j+1} = \alpha_{i,j} \frac{1}{q_{i,j+1}} \frac{f(\mathrm{p}, \omega, \omega')|\cos\theta'|}{p(\omega')}. \quad [15.17]$$

The code here uses uniform random numbers for these samples, since the advantages of further low-discrepancy points are mostly lost with increasing numbers of bounces: the variation in paths introduced by complex scene geometry overwhelms any well-distributed properties of the sample values.

⟨*Sample new photon ray direction*⟩ ≡ 808
```
Vector wi;
float pdf;
BxDFType flags;
Spectrum fr = photonBSDF->Sample_f(wo, &wi, BSDFSample(rng),
                                   &pdf, BSDF_ALL, &flags);
if (fr.IsBlack() || pdf == 0.f) break;
Spectrum anew = alpha * fr *
    AbsDot(wi, photonBSDF->dgShading.nn) / pdf;
⟨Possibly terminate photon path with Russian roulette 811⟩
specularPath &= ((flags & BSDF_SPECULAR) != 0);
```

AbsDot() 61
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection::dg 186
Intersection::rayEpsilon 186
Photon 805
RayDifferential 69
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

```
if (indirectDone && !specularPath) break;
photonRay = RayDifferential(photonIsect.dg.p, wi, photonRay,
                            photonIsect.rayEpsilon);
```

The photon scattering step needs to be implemented carefully in order to keep the photon weights as similar to each other as possible. A method that gives distribution of photons where all have exactly equal weights was suggested by Jensen (2001, Section 5.2). First, the reflectance is computed at the intersection point. A random decision is then made whether or not to continue the photon's path with probability proportional to this reflectance. If the photon continues, its scattered direction is found by sampling from the BSDF's distribution, but it continues with its weight unchanged except for adjusting the spectral distribution based on the surface's color. Thus, a surface that reflects very little light will reflect few of the photons that reach it, but those that are scattered will continue on with unchanged contributions and so forth.

This particular approach isn't possible in `pbrt` due to a subtle implementation detail (a similar issue held for light source sampling previously as well): in `pbrt`, the `BxDF` interfaces are written so that the distribution used for importance sampling BSDFs doesn't necessarily have to perfectly match the actual distribution of the function being sampled. It is all the better if it does, but for many complex BSDFs exactly sampling from its distribution is difficult or impossible.

Therefore, here we will use an approach that generally leads to a similar result but offers more flexibility: at each intersection point, an outgoing direction is sampled with the BSDF's sampling distribution, and the photon's updated weight $\alpha_{i,j+1}$ is computed using Equation (15.17). Then, the ratio of the luminance of $\alpha_{i,j+1}$ to the luminance of the photon's old weight $\alpha_{i,j}$ is used to set the probability of continuing the path after applying Russian roulette.

The probability is thus set so that if the photon's weight is significantly decreased at the scattering point, the termination probability will be high and if the photon's weight is essentially unchanged, the termination probability is low. In particular, the termination probability is chosen in a way such that if the photon continues, after its weight has been adjusted for the possibility of termination, its luminance will be the same as it was before scattering. It is easy to verify this property from the fragment below. (This property actually doesn't hold for the case where $\alpha_{i,j+1} > \alpha_{i,j}$, as can happen when the ratio of the BSDF's value and the PDF is greater than one.)

⟨*Possibly terminate photon path with Russian roulette*⟩ ≡ 810

```
float continueProb = min(1.f, anew.y() / alpha.y());
if (rng.RandomFloat() > continueProb)
    break;
alpha = anew / continueProb;
```

PhotonIntegrator 802
RNG::RandomFloat() 1003
Spectrum::y() 273

Updating the Shared Data Structures

After `blockSize` photon paths have been followed, the task acquires the mutex so that it can safely modify the shared photon map data structures in the `PhotonIntegrator`. The shared `nshot` value is updated to reflect the additional paths followed by this task, and the photons stored in the local arrays are copied into the shared arrays.

⟨*Merge local photon data with data in* PhotonIntegrator⟩ ≡ 806
```
{ MutexLock lock(mutex);
⟨Give up if we're not storing enough photons⟩
nshot += blockSize;
⟨Merge indirect photons into shared array 812⟩
⟨Merge direct, caustic, and radiance photons into shared array⟩
}
```

The task may find that it and the other tasks have generated many paths while storing few to no photons of some types. In this case, it gives up and signals the other tasks to exit by setting `abortTasks` to `true`. (And, if it detects that another task has set this variable, it exits immediately.) This situation may happen, for example, if it was trying to populate a caustic map but there weren't any specular objects in the scene to create caustic paths. This task is handled by the ⟨*Give up if we're not storing enough photons*⟩ fragment, which is straightforward and not included here.

In addition to the photons themselves, the integrator needs to know many paths from the lights were followed to generate the collections of stored photons. These values are necessary for the density estimation algorithm used to interpolate among photons around a point being shaded; they're stored in the `nCausticPaths` and `nIndirectPaths` `PhotonIntegrator` member variables.

⟨*PhotonIntegrator Private Data*⟩ +≡
```
int nCausticPaths, nIndirectPaths;
```

After adding the photons to the shared array, the implementation also checks to see if the additional photons it has added are enough such that the number of requested photons has been found. If so, it updates its `indirectDone` (or, respectively, `causticDone`) variable to indicate that no more photons of that type are needed. There is a subtlety in the implementation here: only the local variable `indirectDone` and not the shared array size is checked before adding the photons to the shared array, so the shared arrays may already have enough photons in them without the additional ones. The rationale is that, since the task here has already gone through the trouble of generating the photons, it might as well store them rather than just throwing them away if enough have already been found. (The fragments that merge the other types of photons are essentially the same and not included here.)

⟨*Merge indirect photons into shared array*⟩ ≡ 812
```
if (!indirectDone) {
    integrator->nIndirectPaths += blockSize;
    for (uint32_t i = 0; i < localIndirectPhotons.size(); ++i)
        indirectPhotons.push_back(localIndirectPhotons[i]);
    localIndirectPhotons.erase(localIndirectPhotons.begin(),
                               localIndirectPhotons.end());
    if (indirectPhotons.size() >= integrator->nIndirectPhotonsWanted)
        indirectDone = true;
}
```

MutexLock 1039
PhotonIntegrator::
nIndirectPaths 812
PhotonIntegrator::
nIndirectPhotonsWanted 803

Once enough of both types of photons have been stored, the task can exit.

⟨*Exit task if enough photons have been found*⟩ ≡ 806
```
if (indirectDone && causticDone)
    break;
```

After all of the photon shooting tasks exit, the `PhotonIntegrator::Preprocess()` method continues execution. It next creates a kd-tree for each of the three photon maps. (The `KdTree` template class used is described in Section A.8 of Appendix A.) Because the direct lighting photons are only used during the preprocessing step and not during scene rendering, their kd-tree is stored in a local variable rather than a `PhotonIntegrator` member variable.

⟨*Build kd-trees for indirect and caustic photons*⟩ ≡ 804
```
KdTree<Photon> *directMap = NULL;
if (directPhotons.size() > 0)
    directMap = new KdTree<Photon>(directPhotons);
if (causticPhotons.size() > 0)
    causticMap = new KdTree<Photon>(causticPhotons);
if (indirectPhotons.size() > 0)
    indirectMap = new KdTree<Photon>(indirectPhotons);
```

⟨*PhotonIntegrator Private Data*⟩ +≡
```
KdTree<Photon> *causticMap;
KdTree<Photon> *indirectMap;
```

Precomputed Radiance Values

If final gathering is being performed, a few tens of final gather rays are traced over the hemisphere at each point passed to `PhotonIntegrator::Li()` in order to sample incident indirect illumination. In order to compute the incident radiance arriving at each final gather ray's origin, it is necessary to compute the outgoing radiance at the intersection point of each of the final gather rays. One option would be to use the `LPhoton()` function (defined later in this section), which searches the kd-tree to find the nearest tens or hundreds of photons around the intersection point and then interpolates among them, though the computational overhead of doing this for each of the final gather rays can be substantial.

We can instead take advantage of the fact that outgoing radiance values computed with the photon map at the final gather ray intersection points don't need to be as accurate as the radiance values computed with the photon map for directly visible points. The `PhotonIntegrator` selects a subset of the initial photons and creates a separate `RadiancePhoton` for each of them. The `RadiancePhoton` stores an exitant radiance value that is computed with the surface's reflectance and the incident radiance distribution at the photon's position. When the intersection point of a final gather ray is found, the *single* nearest `RadiancePhoton` is found and its radiance value is used to approximate the outgoing radiance at the intersection point. This is substantially more efficient than doing a full photon map lookup and photon interpolation.

Figure 15.21 shows an image of a scene rendered using the radiance photons directly, effectively showing the scene as "seen" by the final gathering rays. Note that the illumination appears as a smoothed but generally accurate approximation to the actual illumination.

KdTree 1029
LPhoton() 831
Photon 805
PhotonIntegrator 802
PhotonIntegrator::
causticMap 813
PhotonIntegrator::
indirectMap 813
PhotonIntegrator::Li() 817
RadiancePhoton 815



Figure 15.21: Rendering with Radiance Photons. Scene rendered directly using radiance values from radiance photons to shade visible surfaces. The radiance photons do a reasonable job of capturing an approximation of the outgoing radiance from surfaces in the scene. When used to compute radiance along the final gathering rays, the error they introduce usually isn't detectable, and the final gathering process is substantially accelerated. *(Model courtesy of Florent Boyer.)*

Each time a photon is stored in `PhotonShootingTask::Run()`, a psuedo-random number is used to decide if a `RadiancePhoton` should be created for it; the fragment below takes care of doing so. The outgoing radiance values for the radiance photons can't be computed until all of the photons have been traced and the photon maps have been created. Therefore, the fragment here also stores the BSDF's reflectance and transmittance for use in computing the reflected radiance at this point later.

⟨*Possibly create radiance photon at photon intersection point*⟩ ≡ 810

```
if (depositedPhoton && integrator->finalGather &&
        rng.RandomFloat() < .125f) {
    Normal n = photonIsect.dg.nn;
    n = Faceforward(n, -photonRay.d);
    localRadiancePhotons.push_back(RadiancePhoton(photonIsect.dg.p, n));
    Spectrum rho_r = photonBSDF->rho(rng, BSDF_ALL_REFLECTION);
    localRpReflectances.push_back(rho_r);
    Spectrum rho_t = photonBSDF->rho(rng, BSDF_ALL_TRANSMISSION);
    localRpTransmittances.push_back(rho_t);
}
```

BSDF::rho() 482
BSDF_ALL_REFLECTION 428
BSDF_ALL_TRANSMISSION 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Faceforward() 66
Intersection::dg 186
Normal 65
PhotonIntegrator::finalGather 803
PhotonShootingTask::Run() 806
RadiancePhoton 815
Ray::d 67
RNG::RandomFloat() 1003
Spectrum 263

The RadiancePhoton structure just needs to store the position, surface normal, and outgoing radiance at the point. The surface normal is needed to distinguish between the two sides of the surface, since the outgoing radiance stored will only be used in the hemisphere around the normal. Typically the outgoing radiance on different sides of a surface will be quite different—distinguishing between the two hemispheres here reduces "light leak" image artifacts.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct RadiancePhoton {
    RadiancePhoton(const Point &pp, const Normal &nn)
        : p(pp), n(nn), Lo(0.f) { }
    Point p;
    Normal n;
    Spectrum Lo;
};
```

The reflectances and transmittances only need to be stored temporarily during the preprocessing step between the time the RadiancePhoton is first created and the time that its outgoing radiance value is computed using the fully populated photon maps. Therefore, these values are stored in temporary arrays declared at the top of the Preprocess() photon shooting function.

⟨*Declare shared variables for photon shooting*⟩ +≡ 804
```
vector<Spectrum> rpReflectances, rpTransmittances;
```

Each PhotonShootingTask maintains a local array of these values; they are merged into the shared values after the mutex is acquired in the fragment ⟨*Merge local photon data with data in* PhotonIntegrator⟩.

⟨*Declare local variables for* PhotonShootingTask⟩ +≡ 806
```
vector<Spectrum> localRpReflectances, localRpTransmittances;
```

After the photon maps maps are all filled, exitant radiance can be computed at the radiance photons. The fragment below executes after the photon shooting tasks have finished and the kd-trees that store the photons have been created. It launches ComputeRadiance Tasks to compute the outgoing radiance at the radiance photons; when these tasks are done, it creates a kd-tree to store them.

⟨*Precompute radiance at a subset of the photons*⟩ ≡ 804
```
if (finalGather && radiancePhotons.size()) {
    ⟨Launch tasks to compute photon radiances⟩
    radianceMap = new KdTree<RadiancePhoton>(radiancePhotons);
}
```

KdTree 1029
Normal 65
PhotonIntegrator:: finalGather 803
PhotonIntegrator:: radianceMap 815
PhotonShootingTask 805
Point 63
RadiancePhoton 815
Spectrum 263

⟨*PhotonIntegrator Private Data*⟩ +≡
```
KdTree<RadiancePhoton> *radianceMap;
```

Each task figures out what range of RadiancePhotons it's responsible for computing outgoing radiance values for and performs the computation for each one. The lookupBuf is temporary storage that will be passed to the photon interpolation routine.

⟨*PhotonIntegrator Method Definitions*⟩ +≡
```
void ComputeRadianceTask::Run() {
    ⟨Compute range of radiance photons to process in task 816⟩
    ClosePhoton *lookupBuf = new ClosePhoton[nLookup];
    for (uint32_t i = rpStart; i < rpEnd; ++i) {
        ⟨Compute radiance for radiance photon i 816⟩
    }
    delete[] lookupBuf;
}
```

The radiance photons are partitioned among the tasks as equally as possible. Because the number of radiance photons and the number of tasks don't usually divide equally, the first `excess` tasks are given one more than `taskSize`. Computing the corresponding starting and ending photon for a particular task, then, must account for how many tasks prior to the current one had one extra.

⟨*Compute range of radiance photons to process in task*⟩ ≡ 816
```
uint32_t taskSize = radiancePhotons.size() / numTasks;
uint32_t excess = radiancePhotons.size() % numTasks;
uint32_t rpStart = min(taskNum, excess) * (taskSize+1) +
                   max(0, (int)taskNum-(int)excess) * taskSize;
uint32_t rpEnd = rpStart + taskSize + (taskNum < excess ? 1 : 0);
```

⟨*Compute radiance for radiance photon* i⟩ ≡ 816
```
RadiancePhoton &rp = radiancePhotons[i];
const Spectrum &rho_r = rpReflectances[i], &rho_t = rpTransmittances[i];
if (!rho_r.IsBlack()) {
    ⟨Accumulate outgoing radiance due to reflected irradiance 817⟩
}
if (!rho_t.IsBlack()) {
    ⟨Accumulate outgoing radiance due to transmitted irradiance⟩
}
```

To compute the reflected radiance at the photon, the product of the reflectance (stored when the `RadiancePhoton` was first created) and the irradiance (computed from the photon maps with the `EPhoton()` function) can be used to find an approximate outgoing radiance value for this point, ignoring directional variation in both the incident radiance distribution and the BSDF.

$$\begin{aligned} L_o(p, \omega_o) &= \int_{\mathcal{H}^2(\mathbf{n})} f_r(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos\theta_i|\, d\omega_i \\ &\approx \frac{1}{\pi}\rho_{hh}(p) \int_{\mathcal{H}^2(\mathbf{n})} L_i(p, \omega_i)\, |\cos\theta_i|\, d\omega_i \\ &= \frac{1}{\pi}\rho_{hh}(p)\, E(p, \mathbf{n}) \end{aligned}$$

ClosePhoton 822
EPhoton() 832
RadiancePhoton 815
Spectrum 263
Spectrum::IsBlack() 265

The `EPhoton()` utility function estimates the incident irradiance at the given point with the given surface normal using the photons in the photon map. It will be defined later

in this section, when we discuss how general measurements are computed using the photons.

⟨*Accumulate outgoing radiance due to reflected irradiance*⟩ ≡ 816

```
Spectrum E = EPhoton(directMap, nDirectPaths, nLookup, lookupBuf,
                     maxDistSquared, rp.p, rp.n) +
             EPhoton(indirectMap, nIndirectPaths, nLookup, lookupBuf,
                     maxDistSquared, rp.p, rp.n) +
             EPhoton(causticMap, nCausticPaths, nLookup, lookupBuf,
                     maxDistSquared, rp.p, rp.n);
rp.Lo += INV_PI * rho_r * E;
```

The process for computing outgoing radiance due to transmittance is similar and not included here.

15.6.4 USING THE PHOTON MAP

The photon mapping algorithm partitions the lighting integrand and uses different techniques to evaluate different parts of it. A number of different partitionings have been proposed, and different scenes with different illumination and reflection characteristics may benefit from others than the one implemented here. The `PhotonIntegrator` splits the BSDF into delta and nondelta components. For the nondelta BSDF components, incident radiance L_i is split into incident direct ($L_{i,d}$), caustic ($L_{i,c}$), and indirect ($L_{i,i}$) illumination:

$$\begin{aligned}
&\int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\quad = \int_{S^2} f_\Delta(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\qquad + \int_{S^2} f_{\neg\Delta}(p, \omega_o, \omega_i) L_{i,d}(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\qquad + \int_{S^2} f_{\neg\Delta}(p, \omega_o, \omega_i) L_{i,c}(p, \omega_i) |\cos\theta_i| \, d\omega_i \\
&\qquad + \int_{S^2} f_{\neg\Delta}(p, \omega_o, \omega_i) L_{i,i}(p, \omega_i) |\cos\theta_i| \, d\omega_i.
\end{aligned} \tag{15.18}$$

The first term, delta BSDF components, is evaluated with regular recursive ray tracing, and the second term, direct lighting, is evaluated with the standard direct lighting algorithms of Section 15.1. The standard approaches to these two components handle them efficiently and accurately. The last two terms are evaluated using the photon maps.

We will elide the complete implementation of the `PhotonIntegrator::Li()` method, which is similar in form to the `Li()` methods of the other integrators and will focus on the fragments that evaluate the last two terms of the partitioning above, ⟨*Compute caustic lighting for photon map integrator*⟩ and ⟨*Compute indirect lighting for photon map integrator*⟩. When these fragments run, direct lighting and recursive ray tracing for the specular BSDF components have already been handled in the omitted code.

The `PhotonIntegrator` always uses the caustic photon map to account for light that has taken one or more specular bounces before hitting a nonspecular surface. These paths are particularly difficult to find when tracing paths starting from the point being

EPhoton() 832
INV_PI 1002
PhotonIntegrator 802
PhotonIntegrator::
maxDistSquared 803
PhotonIntegrator::
nLookup 803
RadiancePhoton::Lo 815
RadiancePhoton::n 815
RadiancePhoton::p 815
Spectrum 263



Figure 15.22: Caustics Cast by Light Focusing through a Complex Glass Object Have Remarkable Patterns. Photon mapping is particularly effective at capturing this effect.

shaded; fortunately, because caustics are focused light, there tend to be enough photons to compute good lighting estimates in areas with caustics. Figure 15.22 shows a caustic cast by the dragon model rendered with the photon map. It does a good job of accounting for this lighting effect.

The `LPhoton()` function, which will be described in Section 15.6.5, computes reflected radiance in the direction `wo` for the given BSDF and photon map. Using this function with the caustic photon map is all we need to include the effect of caustics in the computed radiance value here.

⟨*Compute caustic lighting for photon map integrator*⟩ ≡

```
ClosePhoton *lookupBuf = arena.Alloc<ClosePhoton>(nLookup);
L += LPhoton(causticMap, nCausticPaths, nLookup, lookupBuf, bsdf,
             rng, isect, wo, maxDistSquared);
```

Indirect illumination is usually handled with final gathering, though the indirect photon map can also be used directly via the `LPhoton()` function again.

⟨*Compute indirect lighting for photon map integrator*⟩ ≡

```
if (finalGather && indirectMap != NULL) {
    ⟨Do one-bounce final gather for photon map 819⟩
}
else
    L += LPhoton(indirectMap, nIndirectPaths, nLookup, lookupBuf,
                 bsdf, rng, isect, wo, maxDistSquared);
```

ClosePhoton 822
LPhoton() 831
PhotonIntegrator::causticMap 813
PhotonIntegrator::finalGather 803
PhotonIntegrator::indirectMap 813
PhotonIntegrator::maxDistSquared 803
PhotonIntegrator::nCausticPaths 812
PhotonIntegrator::nIndirectPaths 812
PhotonIntegrator::nLookup 803

Final Gathering

When final gathering is being used to estimate reflected radiance to indirect illumination, we'd like to compute the value of the term

$$\int_{S^2} f_{\neg\Delta}(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_{\mathrm{i,i}}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}$$

from Equation (15.18). As usual, in order to apply Monte Carlo integration for estimating this value, we need to sample incident directions ω_i from a distribution that we hope matches the shape of the integrand. The implementation here uses two sampling distributions: one based on the BSDF and one based on an approximation to the incident radiance function $L_{\mathrm{i,i}}$ that is constructed from the indirect illumination photons around the point p.

Given a sampled direction, the incident radiance $L_{\mathrm{i,i}}(\mathrm{p}, \omega_\mathrm{i})$ is found by tracing a ray to find the closest intersection and computing the outgoing radiance there using the single closest `RadiancePhoton`. Figure 15.23 illustrates the basic concept of final gathering, and Figure 15.24 shows the difference between using it or computing indirect lighting directly with the indirect lighting photons.

If the BSDF only has perfectly specular components, there's no reason to do final gathering. Otherwise, a collection of indirect lighting photons around the lookup point is found, and final gather rays are sampled.

⟨*Do one-bounce final gather for photon map*⟩ ≡ 818

```
BxDFType nonSpecular = BxDFType(BSDF_REFLECTION |
    BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
if (bsdf->NumComponents(nonSpecular) > 0) {
    ⟨Find indirect photons around point for importance sampling 821⟩
    ⟨Copy photon directions to local array 823⟩
    ⟨Use BSDF to do final gathering 824⟩
    ⟨Use nearby photons to do final gathering 826⟩
}
```

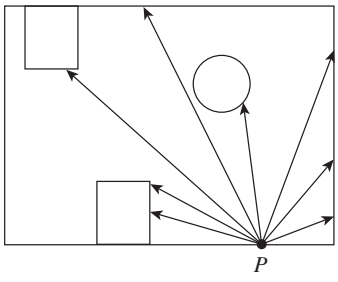


Figure 15.23: The final gathering step with photon mapping traces a set of rays to sample incident radiance at p in order to compute exitant radiance there. However, rather than continuing recursively and tracing more rays at the gather intersection points, the radiance photons are immediately used to compute the outgoing radiance there.

BSDF::NumComponents() 479
BSDF_DIFFUSE 428
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDFType 428
RadiancePhoton 815



(a) (b) (c)

Figure 15.24: Scene (a) with direct illumination only, (b) with photon mapping but without final gathering, and (c) with final gathering. Note that the blurriness from using the photon map directly isn't a problem when final gathering is used. *(Model courtesy of Florent Boyer.)*

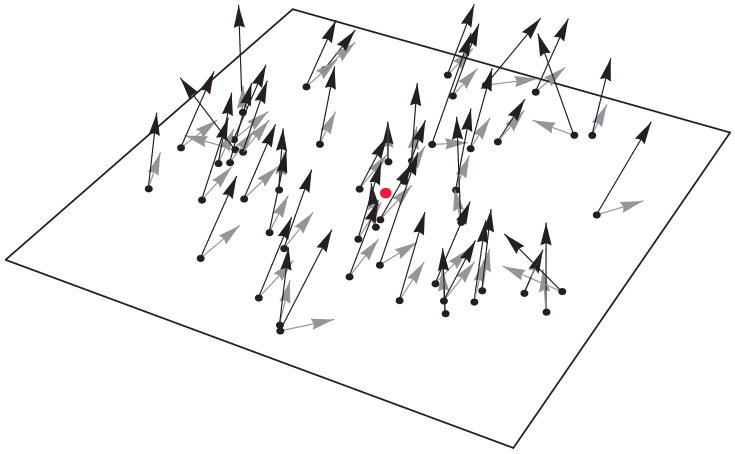


Figure 15.25: Incident directions of 50 photons around a point on the floor of the example scene. Many of the photons are arriving from a cluster of nearby directions. These directions can be used to define a distribution for importance sampling that approximates the distribution of indirect illumination at the point.

The photons around a point in the scene are not just useful for estimating the incident radiance distribution at the point in order to compute the outgoing reflected radiance: in their incident directions, they also carry useful information about the directional distribution of illumination at the point. This information can be put to use to define a sampling distribution for the final gathering rays that tries to match the distribution of indirect illumination. Figure 15.25 shows a plot of the incident directions of 50 photons around a point on the floor of the example scene; note that most of them have incident directions along a relatively similar set of directions.

When choosing ray directions for final gathering from a point, in some cases, sampling the BSDF will be the most effective strategy (for example, where the BSDF is very glossy and the amount of indirect illumination is similar in all directions); in others, sampling the indirect illumination will be a better approach (for example, when the indirect illumination is only from a small subset of directions, and the BSDF is diffuse). Allocating some samples to each approach works well in general, particularly when multiple importance sampling is used when evaluating the Monte Carlo estimator.

First, a fixed number of photons are searched for around the point being shaded by performing a lookup in the indirect photon map. The search radius is progressively widened until the desired number of photons is found. Because the squared distance parameter passed to the `KdTree::Lookup()` method may be modified during the lookup process, it's necessary to store the current search radius in a separate variable here, `searchDist2`.

⟨*Find indirect photons around point for importance sampling*⟩ ≡ 819
```
const uint32_t nIndirSamplePhotons = 50;
PhotonProcess proc(nIndirSamplePhotons,
                   arena.Alloc<ClosePhoton>(nIndirSamplePhotons));
float searchDist2 = maxDistSquared;
while (proc.nFound < nIndirSamplePhotons) {
    float md2 = searchDist2;
    proc.nFound = 0;
    indirectMap->Lookup(p, proc, md2);
    searchDist2 *= 2.f;
}
```

Like the octree used for the irradiance cache, the `KdTree` used to store photons here takes a parameter to its `Lookup()` method that provides an object with a particular method that is called for every item in the search region. `PhotonProcess` handles that role, storing information about the nearby photons that have been passed to it using the `ClosePhoton` structure.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct PhotonProcess {
    ⟨PhotonProcess Public Methods⟩
    ClosePhoton *photons;
    uint32_t nLookup, nFound;
};
```

`ClosePhoton` 822
`KdTree::Lookup()` 1032
`MemoryArena::Alloc()` 1016
`PhotonIntegrator::maxDistSquared` 803
`PhotonProcess` 821
`PhotonProcess::nFound` 821

`ClosePhoton` represents a photon close to the lookup point. To keep track of a photon close to the lookup point, it is only necessary to store a pointer to it. However, the `ClosePhoton` structure also caches the squared distance from the photon to the lookup point in order to more quickly be able to discard the farthest away photon when a closer one is found.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct ClosePhoton {
    ⟨ClosePhoton Public Methods 822⟩
    const Photon *photon;
    float distanceSquared;
};
```

⟨*ClosePhoton Public Methods*⟩ ≡ 822
```
ClosePhoton(const Photon *p = NULL, float md2 = INFINITY)
    : photon(p), distanceSquared(md2) { }
```

`ClosePhoton` also provides an ordering relation, so that the farthest away photon in a collection of photons can be found.

⟨*ClosePhoton Public Methods*⟩ +≡ 822
```
bool operator<(const ClosePhoton &p2) const {
    return distanceSquared == p2.distanceSquared ?
        (photon < p2.photon) : (distanceSquared < p2.distanceSquared);
}
```

As the `KdTree` traverses the tree nodes, it calls the `operator()` method of `PhotonProcess` for each photon inside the search radius. This method stores a pointer to each such photon with the `ClosePhotons` array, `photons`.

⟨*PhotonIntegrator Local Definitions*⟩ ≡
```
inline void PhotonProcess::operator()(const Point &p,
        const Photon &photon, float distSquared, float &maxDistSquared) {
    if (nFound < nLookup) {
        ⟨Add photon to unordered array of photons 823⟩
    }
    else {
        ⟨Remove most distant photon from heap and add new photon 823⟩
    }
}
```

Until `nLookup` photons have been found around the point, this method stores the nearby photons in an unordered array that is filled in whatever order photons are passed to this callback method. However, once the `nLookup`+1st photon arrives (if it does), it is necessary to discard the photon that is farthest away and keep only the `nLookup` closest ones. Therefore, at that point, the `photons` array is reordered to be a heap, such that the root element is the one with the greatest distance from the lookup point. Recall that a heap data structure can be constructed in linear time and that it can be updated after an item is removed or added in logarithmic time. It is much more efficient to organize photons with a heap than to keep them sorted by distance. The `make_heap()` function in the C++ standard library reorders a given array into a heap so that the zeroth element is the root of the heap.

Furthermore, once `nLookup` photons have been found, the search radius that the `KdTree` uses as it traverses its nodes can be decreased; there's no reason to consider any additional

ClosePhoton 822
ClosePhoton::distanceSquared 822
ClosePhoton::photon 822
INFINITY 1002
Photon 805
PhotonProcess 821
PhotonProcess::nFound 821
PhotonProcess::nLookup 821
Point 63

photons that are farther away than the farthest one in the heap. `KdTree::Lookup()` passes a reference to `maxDistSquared` into this callback method so that the callback method can reduce its value in situations like this.

⟨*Add photon to unordered array of photons*⟩ ≡ 822
```
photons[nFound++] = ClosePhoton(&photon, distSquared);
if (nFound == nLookup) {
    std::make_heap(&photons[0], &photons[nLookup]);
    maxDistSquared = photons[0].distanceSquared;
}
```

As additional photons come in after the heap has been built, we know that the squared distance to new ones must be less than the `maxDistSquared`, since the `KdTree` doesn't call the callback method for items that are farther away. Thus, any additional photon must be closer than the root node of the heap, which is the farthest away one that is stored. Therefore, the method immediately calls the standard library routine that removes the root item of the heap and updates the order of the remaining ones in the heap to reestablish a valid heap, `pop_heap()`. The new photon is added to the end of the array, which will have been left empty after `pop_heap()` was called and then `push_heap()` again rebuilds the heap, accounting for a new element added to the end of it. After all of this, `maxDistSquared` can again be reduced to whatever the distance is to the new farthest away photon.

⟨*Remove most distant photon from heap and add new photon*⟩ ≡ 822
```
std::pop_heap(&photons[0], &photons[nLookup]);
photons[nLookup-1] = ClosePhoton(&photon, distSquared);
std::push_heap(&photons[0], &photons[nLookup]);
maxDistSquared = photons[0].distanceSquared;
```

Once the desired number of photons has been found, their directions are copied to a contiguous array. This extra step improves performance for the rest of the final gathering work by ensuring that all of the photons' directions can be found without incurring excessive cache misses. If instead the original `Photon` structures pointed to by `proc.photons[].photon` were used, the pattern of memory accesses would be relatively incoherent and unused extra data from the `Photon` structure would potentially be repeatedly loaded into the cache.

⟨*Copy photon directions to local array*⟩ ≡ 819
```
Vector *photonDirs = arena.Alloc<Vector>(nIndirSamplePhotons);
for (uint32_t i = 0; i < nIndirSamplePhotons; ++i)
    photonDirs[i] = proc.photons[i].photon->wi;
```

Once the incident directions of the indirect photons around the point are available, the final gathering process can begin. In the first stage, samples are taken from the BSDF to determine outgoing directions.

ClosePhoton 822
ClosePhoton::
distanceSquared 822
MemoryArena::Alloc() 1016
Photon 805
Photon::wi 805
PhotonProcess::nFound 821
PhotonProcess::nLookup 821
PhotonProcess::photons 821
Vector 57

⟨*Use BSDF to do final gathering*⟩ ≡ 819
```
Spectrum Li = 0.;
for (int i = 0; i < gatherSamples; ++i) {
    ⟨Sample random direction from BSDF for final gather ray 824⟩
    ⟨Trace BSDF final gather ray and accumulate radiance 824⟩
}
L += Li / gatherSamples;
```

Specular components of the BSDF are ignored when choosing these samples, since they are handled separately with recursive ray tracing.

⟨*Sample random direction from BSDF for final gather ray*⟩ ≡ 824
```
Vector wi;
float pdf;
BSDFSample bsdfSample(sample, bsdfGatherSampleOffsets, i);
Spectrum fr = bsdf->Sample_f(wo, &wi, bsdfSample,
                             &pdf, BxDFType(BSDF_ALL & ~BSDF_SPECULAR));
if (fr.IsBlack() || pdf == 0.f) continue;
```

If an intersection is found for the final gather ray, then the outgoing radiance at the intersection is found using the radiance photons, and a weight for the sample is computed with multiple importance sampling. Since both the BSDF and the indirect lighting distribution approximated by the nearby photons may match important components of the integrand, MIS improves the quality of the final results in cases where one or the other of the distributions matches the integrand much better than the other one.

⟨*Trace BSDF final gather ray and accumulate radiance*⟩ ≡ 824
```
RayDifferential bounceRay(p, wi, ray, isect.rayEpsilon);
Intersection gatherIsect;
if (scene->Intersect(bounceRay, &gatherIsect)) {
    ⟨Compute exitant radiance Lindir using radiance photons 824⟩
    ⟨Compute MIS weight for BSDF-sampled gather ray 825⟩
    Li += fr * Lindir * (AbsDot(wi, n) * wt / pdf);
}
```

Given a lookup point and its surface normal, an instance of the `RadiancePhotonProcess` structure is passed to the `KdTree::Lookup()` method. It stores a pointer to the single closest `RadiancePhoton` to the lookup point. Its contribution is then attenuated by participating media, if any.

⟨*Compute exitant radiance* `Lindir` *using radiance photons*⟩ ≡ 824, 827
```
Spectrum Lindir = 0.f;
Normal nGather = gatherIsect.dg.nn;
nGather = Faceforward(nGather, -bounceRay.d);
RadiancePhotonProcess proc(nGather);
float md2 = INFINITY;
radianceMap->Lookup(gatherIsect.dg.p, proc, md2);
if (proc.photon != NULL)
    Lindir = proc.photon->Lo;
Lindir *= renderer->Transmittance(scene, bounceRay, NULL, rng, arena);
```

AbsDot() 61
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Faceforward() 66
INFINITY 1002
Intersection 186
Intersection::dg 186
Intersection::rayEpsilon 186
KdTree::Lookup() 1032
Normal 65
PhotonIntegrator::
bsdfGatherSampleOffsets 803
RadiancePhoton 815
RadiancePhoton::Lo 815
RadiancePhotonProcess 825
RadiancePhotonProcess::
photon 825
Ray::d 67
RayDifferential 69
Renderer::Transmittance() 25
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57

The RadiancePhotonProcess structure only needs to record the single closest radiance photon to the lookup point, so the photon pointer suffices for this. Because radiance photons only store illumination in a hemisphere around their surface normal, it also needs to store the surface normal to ensure that the surface normal here and the normal of radiance photon used are in the same hemisphere.

⟨*PhotonIntegrator Local Declarations*⟩ +≡
```
struct RadiancePhotonProcess {
    ⟨RadiancePhotonProcess Methods 825⟩
    const Normal &n;
    const RadiancePhoton *photon;
};
```

Along the lines of PhotonProcess, operator() of the RadiancePhotonProcess structure is called by the kd-traversal method for each photon within the search radius. Because it is only necessary to record the single closest photon to the lookup point, the implementation here is particularly straightforward. In looks for the single closest radiance photon with normal pointing in the same hemisphere as the normal at the point the final gather ray intersected a surface, storing a pointer to it and reducing the maximum search radius when it finds one. (Recall that KdTree::Lookup() doesn't call this method for any items outside the search radius, so it's not necessary to check that the given RadiancePhoton is the closest one here.)

⟨*RadiancePhotonProcess Methods*⟩ ≡ 825
```
void operator()(const Point &p, const RadiancePhoton &rp,
                float distSquared, float &maxDistSquared) {
    if (Dot(rp.n, n) > 0) {
        photon = &rp;
        maxDistSquared = distSquared;
    }
}
```

To compute the multiple importance sampling weight, we need to compute the value of the PDF for sampling the direction wi from the photons' distribution. The fragment that does this, ⟨*Compute PDF for photon-sampling of direction* wi⟩, will be defined shortly, after that sampling approach used is explained.

⟨*Compute MIS weight for BSDF-sampled gather ray*⟩ ≡ 824
```
⟨Compute PDF for photon-sampling of direction wi 827⟩
float wt = PowerHeuristic(gatherSamples, pdf, gatherSamples, photonPdf);
```

In an effort to generate samples according to the incident radiance distribution, the sampling approach here takes the photons around the point being shaded and defines a cone of directions centered about each photon's incident direction. The sampling method then chooses one of the photons at random and randomly samples a direction inside its cone to select a final gather ray.

Note that this sampling approach may well never generate samples for some directions ω_i where the amount of indirect illumination is nonzero. Therefore, this wouldn't be an acceptable sampling technique if it was the only method being used with the standard

Normal 65
PhotonIntegrator::
gatherSamples 803
PhotonProcess 821
Point 63
PowerHeuristic() 693
RadiancePhoton 815

Monte Carlo estimator, since there must be some probability of generating any sample with nonzero contribution. However, when using multiple importance sampling, it is only necessary that *one* of the sampling methods used have nonzero probability of sampling directions where the integrand is nonzero. Because this property is required for the BSDF sampling methods, this shortcoming of the photon-based sampling method we have chosen isn't a problem here.

⟨*Use nearby photons to do final gathering*⟩ ≡ 819
```
Li = 0.;
for (int i = 0; i < gatherSamples; ++i) {
    ⟨Sample random direction using photons for final gather ray 826⟩
    ⟨Trace photon-sampled final gather ray and accumulate radiance 827⟩
}
L += Li / gatherSamples;
```

This method first chooses a photon with uniform probability from the collection of nearby photons. (If photon weights were expected to vary significantly, we might want to instead construct a photon sampling distribution based on the photon weights.) Here, we are slightly abusing the semantics of the `BSDFSample` type, using the component sample to choose which photon to use.

⟨*Sample random direction using photons for final gather ray*⟩ ≡ 826
```
BSDFSample gatherSample(sample, indirGatherSampleOffsets, i);
int photonNum = min((int)nIndirSamplePhotons - 1,
    Floor2Int(gatherSample.uComponent * nIndirSamplePhotons));
⟨Sample gather ray direction from photonNum 826⟩
```

Given the photon, a direction is sampled from a cone about its incident direction with uniform probability. This is easily done with the `UniformSampleCone()` routine. Again we are overloading the `BSDFSample` semantics, using its 2D directional sample to sample the direction in the cone. The member variable `cosGatherAngle` is initialized in the `PhotonIntegrator` constructor based on a parameter that can be set in the input file. By default, rays are distributed in a cone that subtends an angle of 10 degrees.

⟨*Sample gather ray direction from* `photonNum`⟩ ≡ 826
```
Vector vx, vy;
CoordinateSystem(photonDirs[photonNum], &vx, &vy);
Vector wi = UniformSampleCone(gatherSample.uDir[0], gatherSample.uDir[1],
                              cosGatherAngle, vx, vy, photonDirs[photonNum]);
```

The same process is used for computing the contributions of final gather rays as was used for rays found by sampling the BSDF; therefore, we reuse the fragment ⟨*Compute exitant radiance* `Lindir` *using radiance photons*⟩ here.

BSDFSample 705
BSDFSample::uComponent 705
BSDFSample::uDir 705
Floor2Int() 1002
PhotonIntegrator::
cosGatherAngle 803
PhotonIntegrator::
indirGatherSampleOffsets 803
UniformSampleCone() 713
Vector 57

⟨*Trace photon-sampled final gather ray and accumulate radiance*⟩ ≡ 826
```
Spectrum fr = bsdf->f(wo, wi);
if (fr.IsBlack()) continue;
RayDifferential bounceRay(p, wi, ray, isect.rayEpsilon);
Intersection gatherIsect;
if (scene->Intersect(bounceRay, &gatherIsect)) {
    ⟨Compute exitant radiance Lindir using radiance photons 824⟩
    ⟨Compute PDF for photon-sampling of direction wi 827⟩
    ⟨Compute MIS weight for photon-sampled gather ray 827⟩
    Li += fr * Lindir * AbsDot(wi, n) * wt / photonPdf;
}
```

It's easy to compute the PDF for sampling a given direction with the nearby photons: it's the average of the PDFs for each of the photons to sample the direction. For each photon, a dot product indicates if the given direction is within the cone of possible directions for that photon; if the direction is inside the cone, the constant PDF for sampling a direction in a cone of the given angle is added to the PDF sum.

A small fudge factor is used in the test for whether a given direction is within a given cone in order to handle the case where `UniformSampleCone()` selected a direction at the very edge of the cone. In that case, the dot product test here might indicate that the direction wasn't in the cone it was sampled from, leading to `photonPdf` being zero and thence to not-a-number or infinite values in the final image.

⟨*Compute PDF for photon-sampling of direction* `wi`⟩ ≡ 825, 827
```
float photonPdf = 0.f;
float conePdf = UniformConePdf(cosGatherAngle);
for (uint32_t j = 0; j < nIndirSamplePhotons; ++j)
    if (Dot(photonDirs[j], wi) > .999f * cosGatherAngle)
        photonPdf += conePdf;
photonPdf /= nIndirSamplePhotons;
```

To find the MIS weight for the direction sampled using the incident photons, the BSDF's PDF for choosing the direction must be computed.

⟨*Compute MIS weight for photon-sampled gather ray*⟩ ≡ 827
```
float bsdfPdf = bsdf->Pdf(wo, wi);
float wt = PowerHeuristic(gatherSamples, photonPdf, gatherSamples, bsdfPdf);
```

AbsDot() 61
BSDF::f() 481
BSDF::Pdf() 708
Dot() 60
Intersection 186
LPhoton() 831
PhotonIntegrator::cosGatherAngle 803
PhotonIntegrator::gatherSamples 803
PowerHeuristic() 693
RayDifferential 69
Scene::Intersect() 23
Spectrum 263
Spectrum::IsBlack() 265
UniformConePdf() 713

15.6.5 PHOTON INTERPOLATION AND DENSITY ESTIMATION

We will now introduce two functions that compute measurements using the photons in the photon maps and connect their implementations to the particle tracing theory introduced in Section 15.6.1. The first function, `LPhoton()`, computes the reflected radiance at a point with a given BSDF due to incident illumination. The second, `EPhoton()`, computes the incident irradiance at a point.

In order to compute reflected radiance at a point, we need to estimate the exitant radiance equation at a point p in a direction ω_o, which can equivalently (and cumbersomely) be

written as a measurement over all points on surfaces in the scene where a Dirac delta distribution selects only particles precisely at p:

$$\int_{\mathbb{S}^2} L_{\mathrm{i}}(\mathrm{p}, \omega_{\mathrm{i}})\ f(\mathrm{p}, \omega_{\mathrm{o}}, \omega_{\mathrm{i}})\ |\cos\theta_{\mathrm{i}}|\ \mathrm{d}\omega_{\mathrm{i}}$$
$$= \int_A \int_{\mathbb{S}^2} \delta(\mathrm{p} - \mathrm{p}') L_{\mathrm{i}}(\mathrm{p}', \omega_{\mathrm{i}})\ f(\mathrm{p}', \omega_{\mathrm{o}}, \omega_{\mathrm{i}})\ |\cos\theta_{\mathrm{i}}|\ \mathrm{d}\omega_{\mathrm{i}}\ \mathrm{d}A(\mathrm{p}'),$$

and so the function that describes the measurement is

$$W_{\mathrm{e}}(\mathrm{p}', \omega) = \delta(\mathrm{p}' - \mathrm{p})\ f(\mathrm{p}, \omega_{\mathrm{o}}, \omega)\ |\cos\theta|.$$

Unfortunately, because there is a delta distribution in W_{e}, all of the particle histories that were generated during the particle-tracing step have zero probability of having nonzero contribution if Equation (15.15) is used to compute the estimate of the measurement value (just as we will never be able to choose a direction from a diffuse surface that intersects a point light source unless the direction is sampled accounting for this).

Here is the point at which bias is introduced into the photon mapping algorithm. Under the assumption that the information about illumination at nearby points can be used to construct an estimate of illumination at the shading point, photon mapping interpolates information about illumination at the point being shaded from nearby photons. The more photons there are around the point and the higher their weights, the more radiance is estimated to be incident at the point. (Recall from the original description of the particle-tracing algorithm and Equation (15.15) that both the local density of the particles and their individual weights together affect their contribution.) This estimated radiance is used in conjunction with the surface's BSDF to compute the reflected light.

A statistical technique called *density estimation* provides the mathematical tools to perform this interpolation. Density estimation constructs a PDF given a set of sample points under the assumption that the samples are distributed according to the overall distribution of some function of interest. Histogramming is a straightforward example of the idea. In 1D, the line is divided into intervals with some width, and one can count how many samples land in each interval and normalize so that the areas of the intervals sum to one.

Kernel methods are a more sophisticated density estimation technique. They generally give better results and smoother PDFs that don't suffer from the discontinuities that histograms do. Given a kernel function $k(x)$ that integrates to one,

$$\int_{-\infty}^{\infty} k(x)\ \mathrm{d}x = 1,$$

the kernel estimator for N samples at locations x_i is

$$\hat{p}(x) = \frac{1}{Nh} \sum_{i=1}^{N} k\left(\frac{x - x_i}{h}\right),$$

where h is the window width (also known as the smoothing parameter). Kernel methods can be thought of as placing a series of bumps at observation points, where the sum of the

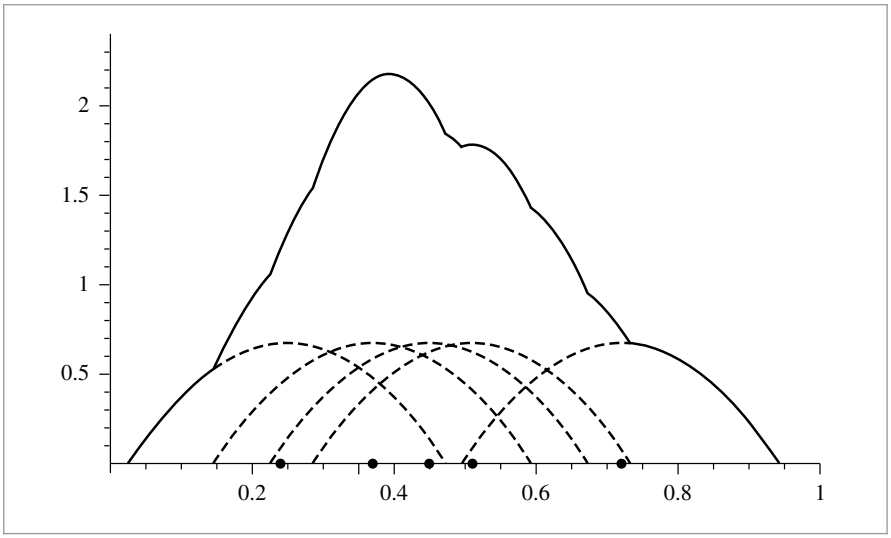


Figure 15.26: 1D example of density estimation, using the Epanechnikov kernel, $k(t) = .75(1 - .2t^2)/\sqrt{5}$, if $t < \sqrt{5}$, 0 otherwise, and a width of 0.1. The points marked with closed circles are the sample points, and an instance of the kernel (dashed lines) is placed over each one. The sum of the kernels gives a properly normalized PDF that attempts to model a distribution that the points could be distributed by.

bumps forms a PDF since they individually integrate to one and the sum is normalized. Figure 15.26 shows an example of density estimation in 1D, where a smooth PDF is computed from a set of sample points.

The key question with kernel methods is how the window width h is chosen. If it is too wide, the PDF will blur out relevant detail in parts of the domain with many samples; if it is too narrow, the PDF will be too bumpy in the tails of the distribution where there aren't many samples. Nearest-neighbor techniques solve this problem by choosing h adaptively based on local density of samples. Where there are many samples, the width is small; where there are few samples, the width is large. For example, one approach is to pick a number n and find the distance to the nth nearest sample from the point x and use that distance, $d_n(x)$, for the window width. This is the *generalized nth nearest-neighbor estimate:*

$$\hat{p}(x) = \frac{1}{N d_n(x)} \sum_{i=1}^{N} k\left(\frac{x - x_i}{d_n(x)}\right).$$

In d dimensions, this generalizes to

$$\hat{p}(x) = \frac{1}{N (d_n(x))^d} \sum_{i=1}^{N} k\left(\frac{x - x_i}{d_n(x)}\right). \qquad [15.19]$$

The implementation here uses Simpson's kernel,

$$k(x) = \begin{cases} \frac{3}{\pi}\left(1 - \left(\frac{d_i(x)}{d_n(x)}\right)^2\right)^2 & |x| < 1 \\ 0 & \text{otherwise.} \end{cases}$$

where $d_i(x)$ is the distance to the ith point used in the density estimate.

Intuitively, it makes sense to use a kernel that gives greater weight to samples near the lookup point and less weight to samples farther away. Doing so is generally a good idea, since the far-away samples give less useful information about the function than the nearby ones. Using this kernel for photon mapping thus also smoothes out the illumination estimates computed by the photon map, a generally desirable property (sharp caustics aside).

The implementation here assumes that no photons with squared distance from the point greater than `maxDist2` will be passed to the kernel function.

⟨*PhotonIntegrator Local Definitions*⟩ +≡

```
inline float kernel(const Photon *photon, const Point &p,
                    float maxDist2) {
    float s = (1.f - DistanceSquared(photon->p, p) / maxDist2);
    return 3.f * INV_PI * s * s;
}
```

Substituting into the measurement equation, it can be shown that the appropriate estimator for the measurement we'd like to compute, the exitant radiance at the point p in direction ω, is given by

$$L_o(\mathrm{p}, \omega_o) \approx \sum_j^N \hat{p}(\mathrm{p}_j)\, \alpha_j\, f(\mathrm{p}, \omega_o, \omega_j), \qquad [15.20]$$

where the sum is over all of the photons and scale factors for the photons are computed based on the density estimation, Equation (15.19). Because we know that the kernel function is zero for points farther away than the nth nearest neighbor distance $d_n(x)$, implementations of this sum only need to sum over the n closest neighbors.

The error introduced by this interpolation can be difficult to quantify. Storing more photons, so that it isn't necessary to use photons as far away, will almost always improve the results, but in general, error will depend on how quickly the illumination is changing at the point. One can always construct pathological cases where this error is unacceptable, but in practice it usually isn't too bad. Because the interpolation step tends to blur out illumination, high-frequency changes in lighting are sometimes poorly reconstructed with photon mapping. Since indirect illumination tends to be low frequency, this isn't too much of a problem in practice. And, although caustics do often have high-frequency variation, photon mapping also works well for them since many photons are focused in caustic regions.

Given this background, the `LPhoton()` function has two main tasks: First, it needs to find the `nLookup` nearest photons to the point where the shading is being done. Second, it needs to use those photons to compute the reflected radiance at the point. The function

DistanceSquared() 65
INV_PI 1002
LPhoton() 831
Photon 805
Photon::p 805
Point 63

starts by checking if the BSDF has any nonspecular components—if it is purely specular, there's no need to do the lookup, since the BSDF won't return any contribution for any of the photons' incident directions.

```
⟨PhotonIntegrator Local Definitions⟩ +≡
  Spectrum LPhoton(KdTree<Photon> *map, int nPaths, int nLookup,
        ClosePhoton *lookupBuf, BSDF *bsdf, RNG &rng,
        const Intersection &isect, const Vector &wo, float maxDist2) {
      Spectrum L(0.);
      BxDFType nonSpecular = BxDFType(BSDF_REFLECTION |
          BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
      if (map && bsdf->NumComponents(nonSpecular) > 0) {
          ⟨Do photon map lookup at intersection point 831⟩
          ⟨Estimate reflected radiance due to incident photons 831⟩
      }
      return L;
  }
```

The previously introduced `PhotonProcess` object and the `KdTree` take care of finding the `nLookup` nearest neighbors.

```
⟨Do photon map lookup at intersection point⟩ ≡ 831
  PhotonProcess proc(nLookup, lookupBuf);
  map->Lookup(isect.dg.p, proc, maxDist2);
```

Once the photons have been found, the value of Equation (15.20) can be computed. If the surface is purely diffuse, a separate code path is used, since those surfaces can be handled more efficiently.

```
⟨Estimate reflected radiance due to incident photons⟩ ≡ 831
  ClosePhoton *photons = proc.photons;
  int nFound = proc.nFound;
  Normal Nf = Faceforward(bsdf->dgShading.nn, wo);
  if (bsdf->NumComponents(BxDFType(BSDF_REFLECTION |
          BSDF_TRANSMISSION | BSDF_GLOSSY)) > 0) {
      ⟨Compute exitant radiance from photons for glossy surface 832⟩
  }
  else {
      ⟨Compute exitant radiance from photons for diffuse surface⟩
  }
```

If the BSDF has any nondiffuse components, `LPhoton()` loops over the nearby photons to evaluate the sum in Equation (15.20).

BSDF 478
BSDF_DIFFUSE 428
BSDF_GLOSSY 428
BSDF_REFLECTION 428
BSDF_TRANSMISSION 428
BxDFType 428
ClosePhoton 822
ClosePhoton::photon 822
DifferentialGeometry::p 102
Faceforward() 66
Intersection 186
Intersection::dg 186
KdTree 1029
KdTree::Lookup() 1032
Normal 65
Photon 805
PhotonIntegrator::
 nLookup 803
PhotonProcess 821
RNG 1003
Spectrum 263
Vector 57

⟨*Compute exitant radiance from photons for glossy surface*⟩ ≡ 831
```
for (int i = 0; i < nFound; ++i) {
    const Photon *p = photons[i].photon;
    float k = kernel(p, isect.dg.p, maxDist2);
    L += (k / (nPaths * maxDist2)) * bsdf->f(wo, p->wi) *
         p->alpha;
}
```

For purely diffuse BSDFs, it's wasteful to call the `BSDF::f()` method `nFound` times, since it will always return a constant value. In this case, the equivalent computation can be done much more efficiently by finding the weighted sum of photon α values and multiplying it by the constant BSDF, found by dividing the surface's reflectance by π. We won't include the fragment that does this, ⟨*Compute exitant radiance from photons for diffuse surface*⟩, since its implementation is straightforward.

Following a similar approach, the `EPhoton()` function estimates incident irradiance at a given point about the hemisphere with a given normal using the provided photon map.

⟨*PhotonIntegrator Local Definitions*⟩ +≡
```
Spectrum EPhoton(KdTree<Photon> *map, int count, int nLookup,
        ClosePhoton *lookupBuf, float maxDist2, const Point &p,
        const Normal &n) {
    if (!map) return 0.f;
    ⟨Lookup nearby photons at irradiance computation point 832⟩
    ⟨Accumulate irradiance value from nearby photons 833⟩
}
```

This method first performs the usual photon map lookup, finding the given number of photons around the lookup point.

⟨*Lookup nearby photons at irradiance computation point*⟩ ≡ 832
```
PhotonProcess proc(nLookup, lookupBuf);
float md2 = maxDist2;
map->Lookup(p, proc, md2);
```

Given these photons, density estimation is again performed, this time to estimate the incident irradiance at the point. How to compute the value of this estimate can be determined by defining a measurement function to compute irradiance at a point p with normal **n**:

$$W_e(\mathrm{p}', \omega) = \delta(\mathrm{p}' - \mathrm{p}) \begin{cases} 1 & (\mathbf{n} \cdot \omega) > 0 \\ 0 & \text{otherwise,} \end{cases}$$

and following the same approach used for `LPhoton()` to derive how to use photons to compute reflected radiance using density estimation. Note that photons in the opposite hemisphere from the given surface normal are ignored, as they don't contribute to the outgoing radiance in the hemisphere of interest. Here, also, a constant kernel is used since the irradiance values are only used for the radiance photons, which aren't seen directly.

BSDF::f() 481
ClosePhoton 822
ClosePhoton::photon 822
DifferentialGeometry::p 102
EPhoton() 832
Intersection::dg 186
KdTree 1029
KdTree::Lookup() 1032
LPhoton() 831
Normal 65
Photon 805
Photon::alpha 805
PhotonProcess 821
Point 63
Spectrum 263

⟨*Accumulate irradiance value from nearby photons*⟩ ≡ 832

```
if (proc.nFound == 0) return Spectrum(0.f);
ClosePhoton *photons = proc.photons;
Spectrum E(0.);
for (uint32_t i = 0; i < proc.nFound; ++i)
    if (Dot(n, photons[i].photon->wi) > 0.)
        E += photons[i].photon->alpha;
return E / (count * md2 * M_PI);
```

*15.7 METROPOLIS LIGHT TRANSPORT

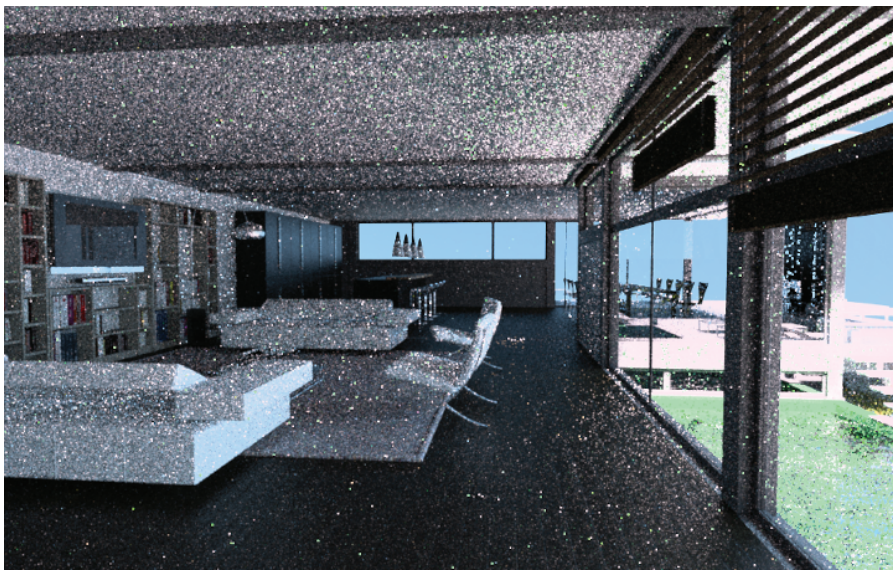
This chapter concludes with a `Renderer` that uses Metropolis sampling (first introduced in Section 13.4) as the foundation of an unbiased method for rendering images, Metropolis Light Transport (MLT).

MLT generates a sequence of light-carrying paths through the scene, where each path is found by mutating the previous path in some manner. These mutations are done in a way that ensures that the overall distribution of sampled paths is proportional to the contribution the paths make to the image being generated. Such a distribution of paths can in turn be used to generate an image of the scene. Given the flexibility of the Metropolis sampling method, there are relatively few restrictions on the types of mutations that can be applied; in general, it is possible to apply unusual sampling techniques designed to sample otherwise difficult-to-find light-carrying paths. Many such sampling techniques that can be used with MLT can't easily be applied to other Monte Carlo light transport algorithms without introducing bias.

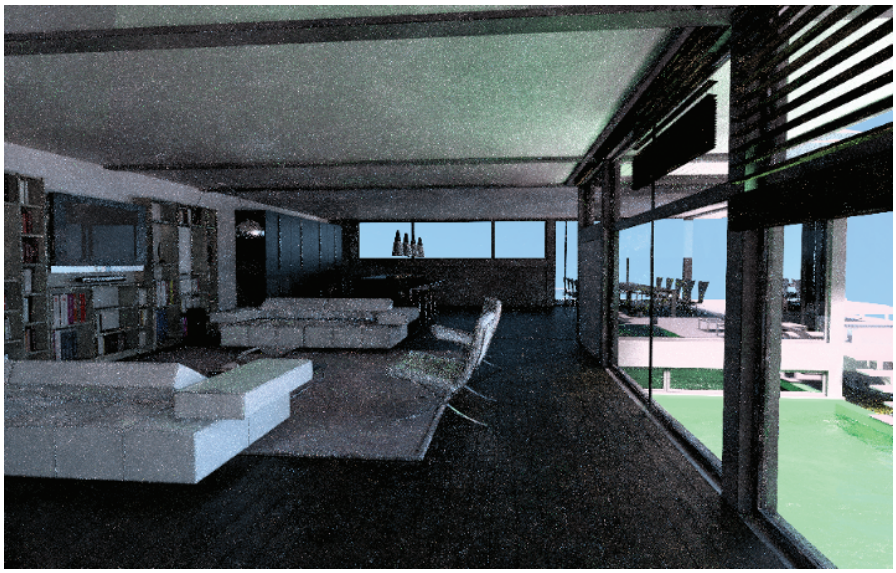
MLT has another important advantage with respect to previous unbiased approaches for image synthesis in that it performs *local exploration*: when a path that makes a large contribution to the image is found, it's easy to sample other paths that are similar to that one by making small perturbations to it. When a function has a small value over most of its domain and a large contribution in only a small subset of it, local exploration amortizes the expense (in samples) of the search for the important region by taking a number of samples from this part of the path space. This property makes MLT a reasonably robust light transport algorithm: while it is about as efficient as other unbiased techniques (e.g., path tracing or bidirectional ray tracing) for relatively straightforward lighting problems, it distinguishes itself in more difficult settings where most of the light transport happens along a small fraction of all of the possible paths through the scene.

Figure 15.27 shows the contemporary house scene rendered with path tracing and MLT, with an equal number of samples taken for each approach, and Figure 15.28 compares them with the San Miguel scene. For both, MLT generates a better result, but the difference is particularly pronounced with the house scene, which is a particularly challenging scene for light transport algorithms in that there is essentially no direct illumination inside the house; all light-carrying paths must follow specular bounces through the glass windows. Table 15.1 helps illustrate why this is so: for both of these scenes, regular path tracing has a lot of trouble finding paths that carry any radiance, while Metropolis is more effective at doing so thanks to path reuse.

ClosePhoton 822
ClosePhoton::photon 822
M_PI 1002
Photon::alpha 805
Photon::wi 805
PhotonProcess::nFound 821
PhotonProcess::photons 821
Renderer 24
Spectrum 263

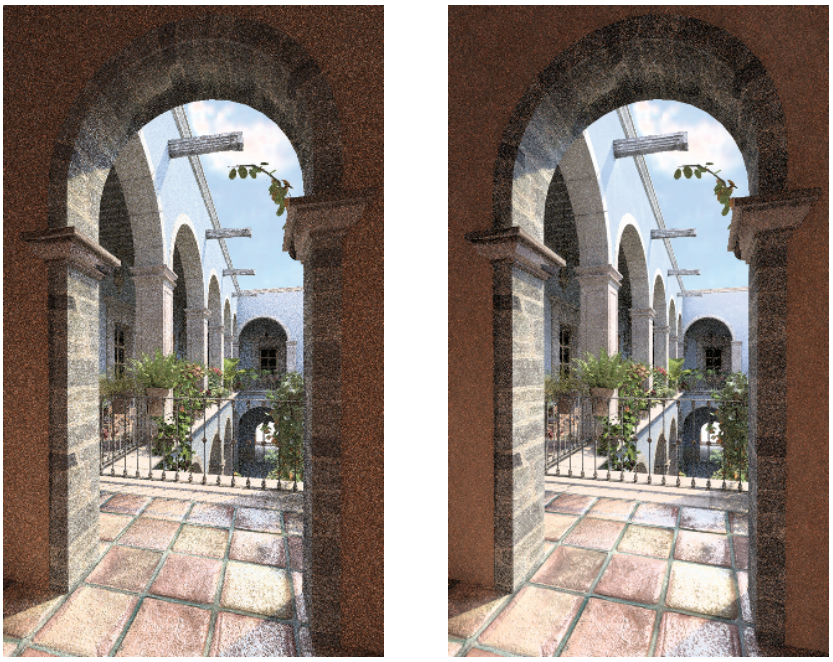


(a)



(b)

Figure 15.27: Comparison of Path Tracing and Metropolis Light Transport. (a) Rendered with path tracing with 256 samples per pixel. Even with many samples, the image is quite noisy. (b) Rendered with Metropolis Light Transport with an average of 256 mutations per pixel. MLT generates a substantially better image for the same amount of work. Path tracing does do better in a few localized parts of the image: note the back wall underneath the windows, the bevel around the television, and the wooden blinds by the window on the upper right, for example. Each of these is a relatively dark region of the image; the fixed sampling rate of the path tracing integrator gives them more samples than Metropolis, which adapts the sampling rate to do more work in brighter areas. *(Model courtesy of Florent Boyer.)*



(a) (b)

Figure 15.28: Comparison of Path Tracing and Metropolis Light Transport. (a) Rendered with path tracing with 256 samples per pixel. (b) Rendered with Metropolis Light Transport with an average of 256 mutations per pixel. As with Figure 15.27, MLT is more effective, but, because this is a less difficult scene in which to find light transport paths, the difference is less striking here. *(Model courtesy of Guillermo M. Leal Llaguno.)*

Table 15.1: Percentage of Traced Paths That Carried Zero Radiance. With these two scenes, path tracing has trouble finding light-carrying paths: only one in 10 or one in 20 ray paths traced carries any radiance at all. Thanks to local exploration, Metropolis is better able to find additional light-carrying paths after one has been found. This is one of the reasons why it is more efficient than path tracing here.

	Path Tracing	Metropolis
Modern House	95.8%	53.6%
San Miguel	89.4%	50.6%

In order to see how to apply Metropolis to the light transport problem, recall the image function formalism introduced in Section 7.1.5. First, consider the case of rendering a scene with Whitted ray tracing and no area light sources: in this case, integration at a surface is deterministic and requires no random sampling, so the contribution of a ray is fully determined by its 5D camera sample of image position, time, and lens position. In this case, the image contribution function is $L(x, y, t, u, v)$, where x and y are the image position, t is the ray's time, and u and v give the lens position.

In the more general case of path tracing, L is an infinite-dimensional function $L(X_1, X_2, \ldots)$, where all of the random samples used in integration are represented by elements of X_i.[8] Beyond the camera samples, the additional samples provide values used for BSDF sampling, direct lighting calculation, Russian roulette tests, and so forth. Given a particular light transport algorithm (such as path tracing), we can define a mapping between the samples X_i and the particular sample values used for rendering. For example, the first five X_i values might give the camera sample, the next three might be used for sampling the BSDF to find the reflected direction at the first intersection, and so forth.

With the `SamplerRenderer` and the various `Integrators` in `pbrt`, a `Sampler` generates most of the samples X_i—the camera sample in a `CameraSample` structure, and whichever of the additional samples X_i that the surface and volume integrators have requested. Then, the integrators themselves may generate some of their sample values themselves. (For example, the `PathIntegrator` uses samples from the sampler for the first few bounces, but then just generates uniform random numbers with a `RNG` for subsequent bounces.) In the `MetropolisRenderer` implemented in this section, no `Sampler` at all is used and the renderer is responsible for both generating all of the samples and evaluating their contributions.

Given the formalism of an explicit sample vector and the radiance function, we can apply Metropolis to generating the vectors of samples $(X_1, X_2, \ldots)$ for rendering an image. Recall that Metropolis sampling generates samples according to the distribution of the function being sampled. This is an intuitively desirable property: more sampling effort is naturally placed in parts of the sampling space where more light transport is occurring. The state space Ω for this problem is thus the sample vectors $(X_1, X_2, \ldots) \in [0, 1]^\infty$, where L's value is zero for samples outside of the image's extent.[9] For brevity in the equations below, we will write the state vector as

$$\mathbf{X} = (X_1, X_2, \ldots).$$

Metropolis sampling generates samples from the distribution of a given scalar function, so there are two issues that must be addressed in order to use Metropolis for rendering. First, we need to use these samples to compute integrals of the radiance function, and, second, we need to handle the fact that L is a spectrally valued function but Metropolis needs a scalar function.

We can apply the ideas from Section 13.4.4 to use Metropolis samples to compute integrals. First, we define the *image contribution function* such that for an image with j pixels, each pixel I_j has a value that is the integral of the product of the pixel's image reconstruction filter h_j and the radiance L that contributes to the image:

$$I_j = \int_\Omega h_j(\mathbf{X})\, L(\mathbf{X})\, d\Omega.$$

CameraSample 342
Integrator 740
MetropolisRenderer 852
PathIntegrator 766
RNG 1003
Sampler 340
SamplerRenderer 25

8 Although L is an infinite-dimensional function, only a finite number of samples of X_i will ever be needed due to path termination with Russian roulette.

9 In the implementation to follow, the film samples are scaled to be in the range of pixel coordinates, and time samples are scaled to be in the camera shutter open time range. However, for the mathematical definition of MLT here, we will operate in terms of all samples being in $[0, 1]$.

Generally, the filter function h_j only depends on whichever two samples of $\mathbf{X}$ that give the image sample position. Further, the value of h_j for any particular pixel is usually zero for the vast majority of samples $\mathbf{X}$ due to the filter's finite extent.

If N samples $\mathbf{X}_i$ are generated from some distribution, $\mathbf{X}_i \sim p(\mathbf{X})$, then the standard Monte Carlo estimate of I_j is

$$I_j \approx \frac{1}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{p(\mathbf{X}_i)}.$$

Because L is a spectrally valued function, Metropolis can't be applied directly to generating samples from its distribution; there is no unambiguous notion of what it means to generate samples according to L's distribution. We can, however, define a *scalar contribution function* $I(\mathbf{X})$ to be the function that defines the distribution of samples that are taken with Metropolis sampling. It is desirable that this function be large when L is large so that the distribution of samples has some relationship to the important regions of L. As such, using the luminance of the radiance value is a good choice for the scalar contribution function. In general, any function that is nonzero when L is nonzero will generate correct results, just possibly not as efficiently as a function that is more directly proportional to L.

Given a suitable scalar contribution function, $I(\mathbf{X})$, Metropolis generates a sequence of samples $\mathbf{X}_i$ from I's distribution, the normalized version of I:

$$p(\mathbf{X}) = \frac{I(\mathbf{X})}{\int_\Omega I(\mathbf{X})\, \mathrm{d}\Omega},$$

and the pixel values can thus be computed as

$$I_j \approx \frac{1}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{I(\mathbf{X}_i)} \left(\int_\Omega I(\mathbf{X})\, \mathrm{d}\Omega \right).$$

The integral of I over the entire domain Ω can be computed using a traditional approach like path tracing. If this value is denoted by b, with $b = \int I(\mathbf{X})\, \mathrm{d}\Omega$, then each pixel's value is given by

$$I_j \approx \frac{b}{N} \sum_{i=1}^{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{I(\mathbf{X}_i)}. \tag{15.21}$$

In other words, we can use Metropolis sampling to generate samples $\mathbf{X}_i$ from the distribution of the scalar contribution function I. For each sample, the pixels it contributes to (based on the extent of the pixel filter function h) have the value

$$\frac{b}{N} \frac{h_j(\mathbf{X}_i)\, L(\mathbf{X}_i)}{I(\mathbf{X}_i)}$$

added to them. Thus, brighter pixels have larger values than dimmer pixels due to more samples contributing to them (as long as the ratio $L(\mathbf{X}_i)/I(\mathbf{X}_i)$ is generally of the same magnitude). The `Film::Splat()` method defined in Section 7.8.1 is used to accumulate these contributions.

Film::Splat() 403

The MetropolisRenderer, implemented in the remainder of this section, applies Metropolis sampling and Equation (15.21) to render images, using either standard path tracing or bidirectional path tracing to compute the radiance function L. It is defined in the files renderers/metropolis.h and renderers/metropolis.cpp.

15.7.1 SAMPLE REPRESENTATION

We will first define a representation of the vector of sample values $\mathbf{X}$ used for evaluating the radiance value. While the samples could just be represented as a single vector<float>, the MLT implementation below is easier to understand if the sample vector is refined into sections that are more clearly named.

First, instances of the PathSample structure are used when constructing paths of intersection points on surfaces in the scene (starting either from the camera or from a light). In addition to the sample values needed to sample an outgoing direction with the BSDF, rrSample is available for terminating the path with Russian roulette.

⟨*Metropolis Local Declarations*⟩ ≡
```
struct PathSample {
    BSDFSample bsdfSample;
    float rrSample;
};
```

Similarly, LightingSample has all of the information needed to compute direct illumination at a point with multiple importance sampling. The BSDFSample is used for sampling a direction with the BSDF, and the remaining samples are used for selecting a light source and a point on it. Note that the BSDF sample here is used only for computing direct lighting with multiple importance sampling—it is not used in sampling the sequence of intersection points of a light-carrying path.

⟨*Metropolis Local Declarations*⟩ +≡
```
struct LightingSample {
    BSDFSample bsdfSample;
    float lightNum;
    LightSample lightSample;
};
```

The MLTSample structure puts all of the sample values together; it represents a complete vector of sample values. The CameraSample from Section 7.2.1 is used to compute the initial ray leaving the camera. If path tracing is used to evaluate the radiance function, then the cameraPathSamples are used to generate a path from the camera. At each vertex of the path, the sample values in the corresponding lightingSamples element are used to compute direct lighting at the point.

When bidirectional path tracing is used, lightNumSample is used to select which light source to start the path from and the lightRaySamples are used to generate the initial ray leaving the light. Then, lightPathSamples are used to generate the vertices of the path leaving the light. Figure 15.29 summarizes how the various samples are used in constructing light paths and computing their contributions.

BSDFSample 705
CameraSample 342
LightSample 710
MetropolisRenderer 852

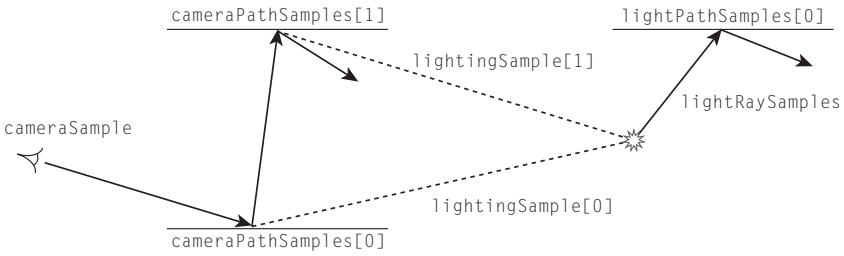


Figure 15.29: Contributions of Members of the `MLTSample` Structure to the Definition of a Path through the Scene. The `cameraSample` defines the camera ray leaving the camera, and the `cameraPathSamples` define how to sample the outgoing directions at the vertices of the camera path. At each camera path vertex, the corresponding `lightingSamples` sample is used for a direct lighting calculation. If bidirectional path tracing is being used, then `lightNumSample` and `lightRaySamples` are used to generate a ray leaving a light source, and `lightPathSamples` are used to sample outgoing directions at light path vertices.

For implementation simplicity, the `MLTSample` allocates storage for samples up to a predetermined maximum path length. To support arbitrary-length light-carrying paths, the lengths of the sample arrays could instead grow on demand and only the samples needed could be generated. See Exercise 15.24 on lazy generation of sample values for MLT for further discussion of this issue.

⟨*Metropolis Local Declarations*⟩ +≡

```
struct MLTSample {
    MLTSample(int maxLength) {
        cameraPathSamples.resize(maxLength);
        lightPathSamples.resize(maxLength);
        lightingSamples.resize(maxLength);
    }
    CameraSample cameraSample;
    float lightNumSample, lightRaySamples[5];
    vector<PathSample> cameraPathSamples, lightPathSamples;
    vector<LightingSample> lightingSamples;
};
```

15.7.2 MUTATIONS

The `MetropolisRenderer` applies two different mutations to the `MLTSample` values. The first (the "large step" mutation) computes new uniform random samples for all of the sample values in **X**. Recall from Section 13.4.2 that it is important that there be greater than zero probability that every possible sample value be proposed; this is taken care of by the large step mutation. In general, the large step mutation helps us widely explore the entire state space without getting stuck on local "islands."

The second mutation (the "small step" mutation) makes a small perturbation to each of the sample values. For difficult lighting configurations, this mutation incrementally

CameraSample 342
LightingSample 838
MetropolisRenderer 852
MLTSample 839
PathSample 838

explores light-carrying paths nearby the current one, so that once a path that makes a significant contribution to the image is found, other paths nearby it can be sampled.

Both of these are symmetric mutations, so their transition probabilities cancel out when the acceptance probability is computed and thus don't need to be computed, as was shown in Equation (13.8).

Implementation of the large step mutation is mostly a matter of calling `RNG::RandomFloat()` to generate new values for all of the samples in the `MLTSample`, with one subtlety related to how new (x, y) image samples are generated, to be explained below.

⟨*Metropolis Local Declarations*⟩ +≡
```
static void LargeStep(RNG &rng, MLTSample *sample, int maxDepth,
        float x, float y, float t0, float t1, bool bidirectional) {
    ⟨Do large step mutation of cameraSample 840⟩
    for (int i = 0; i < maxDepth; ++i) {
        ⟨Apply large step to ith camera PathSample 841⟩
        ⟨Apply large step to ith LightingSample⟩
    }
    if (bidirectional) {
        ⟨Apply large step to bidirectional light samples⟩
    }
}
```

One shortcoming with Metropolis sampling for generating images is that random mutations don't do a good job of ensuring that pixel samples are well stratified over the image plane. In particular, there's no guarantee that all of the pixels will have at least one sample! While the resulting image will still be unbiased in this case, it may be perceptually less pleasing than an image computed with alternative techniques.

To address this issue, `LargeStep()` takes an (x, y) pixel location from the caller; this position is used to set the `imageX` and `imageY` values in the `cameraSample` member variable. It is the caller's responsibility to ensure that in the aggregate, the (x, y) values provided to `LargeStep()` do a good job of sampling the pixels in the image and that they sample the pixels in random order. (For example, if the provided (x, y) values swept across the image in scanline order, taking a single sample in each pixel, the resulting image would be incorrect: that sampling method is not symmetric, while the implementation below expects symmetric transition probabilities.) Ensuring that a minimum set of well-distributed samples are always taken in this manner means that overall image quality improves and a better job is done of (for example) antialiasing geometric edges than would be done otherwise.

⟨*Do large step mutation of* `cameraSample`⟩ ≡ 840
```
sample->cameraSample.imageX = x;
sample->cameraSample.imageY = y;
sample->cameraSample.time = Lerp(rng.RandomFloat(), t0, t1);
sample->cameraSample.lensU = rng.RandomFloat();
sample->cameraSample.lensV = rng.RandomFloat();
```

CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
Lerp() 1000
MLTSample 839
MLTSample::cameraSample 839
RNG 1003
RNG::RandomFloat() 1003

Computing new values for the camera path samples is a matter of looping over the remaining elements in the structure and setting their values with `RNG::RandomFloat()`. We thus won't include implementations of the ⟨*Apply large step to i th* `LightingSample`⟩ ⟨*Apply large step to bidirectional light samples*⟩ fragments here since their implementations are essentially all the same.

⟨*Apply large step to i th camera* `PathSample`⟩ ≡ 840
```
PathSample &cps = sample->cameraPathSamples[i];
cps.bsdfSample.uComponent = rng.RandomFloat();
cps.bsdfSample.uDir[0] = rng.RandomFloat();
cps.bsdfSample.uDir[1] = rng.RandomFloat();
cps.rrSample = rng.RandomFloat();
```

The `SmallStep()` function applies a small mutation to the sample value drawn from an exponential distribution as defined by Equation (13.9): $X_i' = X_i \pm b\, e^{-\log(b/a)\,\xi}$, giving a sample in the range $[a, b]$. The advantage of sampling with an exponential distribution like this is that it naturally tries a variety of mutation sizes. It preferentially makes small mutations, close to the minimum magnitudes specified, which help locally explore the path space in small areas of high contribution where large mutations would tend to be rejected. On the other hand, because it also can make larger mutations, it also avoids spending too much time in a small part of the path space in cases where larger mutations have a good likelihood of acceptance.

The `mutate()` utility function mutates the given value using this approach. One important detail is that if the mutated value is outside the range `[min, max]` it wraps around to the other end of the domain: doing so ensures that the transition probabilities for all pairs of sample values are symmetric.

⟨*Metropolis Local Declarations*⟩ +≡
```
static inline void mutate(RNG &rng, float *v, float min = 0.f,
                          float max = 1.f) {
    if (min == max) { *v = min; return; }
    float a = 1.f / 1024.f, b = 1.f / 64.f;
    static const float logRatio = -logf(b/a);
    float delta = (max - min) * b * expf(logRatio * rng.RandomFloat());
    if (rng.RandomFloat() < 0.5f) {
        *v += delta;
        if (*v > max) *v = min + (*v - max);
    }
    else {
        *v -= delta;
        if (*v < min) *v = max - (min - *v);
    }
}
```

BSDFSample::uComponent 705
BSDFSample::uDir 705
MLTSample::cameraPathSamples 839
PathSample 838
PathSample::bsdfSample 838
PathSample::rrSample 838
RNG 1003
RNG::RandomFloat() 1003

The `SmallStep()` function uses `mutate()` to mutate each of the sample values. Here only the mutation of the camera sample is shown; the remainder of the samples are handled in the ⟨*Apply small step mutation to camera, lighting, and light samples*⟩ fragment, not included here.

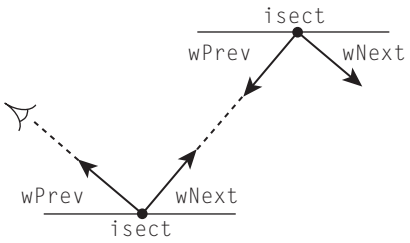


Figure 15.30: Values Stored in the PathVertex at Each Intersection in a Path.

⟨*Metropolis Local Declarations*⟩ +≡
```
static void SmallStep(RNG &rng, MLTSample *sample, int maxDepth,
        int x0, int x1, int y0, int y1, float t0, float t1,
        bool bidirectional) {
    mutate(rng, &sample->cameraSample.imageX, x0, x1);
    mutate(rng, &sample->cameraSample.imageY, y0, y1);
    mutate(rng, &sample->cameraSample.time, t0, t1);
    mutate(rng, &sample->cameraSample.lensU);
    mutate(rng, &sample->cameraSample.lensV);
    ⟨Apply small step mutation to camera, lighting, and light samples⟩
}
```

15.7.3 GENERATING PATHS

Given the few functions that generate and mutate sample vectors above, it's also useful to have a routine that transforms samples into paths through the scene. The `GeneratePath()` function below takes an initial ray (sampled from the camera or a light) and a vector of `PathSample`s that it uses to sample a path through the scene. At each vertex of the path, an instance of the `PathVertex` structure is initialized to store the information about the surface intersection at the corresponding vertex. Figure 15.30 illustrates some of the values stored.

The geometric intersection information for the vertex is stored in the `Intersection`, and normalized vectors to the previous and next vertex in the path are stored in `wPrev` and `wNext`.[10] The `BSDF` at the vertex is stored in `bsdf`. `specularBounce` indicates whether the BSDF component that determined the outgoing direction at this vertex was a delta distribution, and `nSpecularComponents` stores how many specular components the `BSDF` has; the information from these will be needed when computing path contributions with bidirectional path tracing. Finally, the `alpha` member stores the path throughput up to the current vertex (recall the definition of path throughput from Equation (15.7)), scaled by either the emitted light if the path is a light path or the importance from

BSDF 478
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
MLTSample 839
MLTSample::cameraSample 839
mutate() 841
PathSample 838
PathVertex 843
RNG 1003

10 Note that this representation is slightly redundant, in that, when present, the previous vertex's `wNext` vector is equal to the negated `wPrev` vector of a vertex. However, this small redundancy is worth the more straightforward code it makes possible.

the camera (here, the ray weight returned by `Camera::GenerateRayDifferential()`) for camera paths.

⟨*Metropolis Local Declarations*⟩ +≡
```
struct PathVertex {
    Intersection isect;
    Vector wPrev, wNext;
    BSDF *bsdf;
    bool specularBounce;
    int nSpecularComponents;
    Spectrum alpha;
};
```

The `GeneratePath()` method takes a ray and and its initial contribution, either importance W_e from the camera or radiance L_e from the light. It expects that the `PathVertex` pointer passed to it will point to an array of at least `samples.size()` `PathVertex`es; it initializes the vertices and returns the length of the actual path generated. (This length may be less than `samples.size()` if the path is terminated with Russian roulette or if a ray leaves the scene.) If the path does terminate because one of the rays traced along the way didn't intersect any scene geometry, this ray is returned in `*escapedRay` (if non-`NULL`) and the current `alpha` value is returned in `*escapedAlpha`.

⟨*Metropolis Method Definitions*⟩ ≡
```
static uint32_t GeneratePath(const RayDifferential &r,
        const Spectrum &a, const Scene *scene, MemoryArena &arena,
        const vector<PathSample> &samples, PathVertex *path,
        RayDifferential *escapedRay, Spectrum *escapedAlpha) {
    RayDifferential ray = r;
    Spectrum alpha = a;
    if (escapedAlpha) *escapedAlpha = 0.f;
    uint32_t length = 0;
    for (; length < samples.size(); ++length) {
        ⟨Try to generate next vertex of ray path 843⟩
    }
    return length;
}
```

BSDF 478
Intersection 186
MemoryArena 1015
PathSample 838
PathVertex 843
PathVertex::isect 843
RayDifferential 69
Scene 22
Scene::Intersect() 23
Spectrum 263
Vector 57

Each time through the loop, `ray` holds either the provided ray `r` (for the first bounce), or the ray leaving the previous vertex of the path. If this ray doesn't intersect any scene geometry, processing here is done. Otherwise, the `path` array is updated with information about the intersection found before a new ray leaving that intersection is sampled.

⟨*Try to generate next vertex of ray path*⟩ ≡ 843
```
PathVertex &v = path[length];
if (!scene->Intersect(ray, &v.isect)) {
    ⟨Handle ray that leaves the scene during path generation 844⟩
}
⟨Record information for current path vertex 844⟩
⟨Sample direction for outgoing Metropolis path direction 844⟩
⟨Terminate path with RR or prepare for finding next vertex 844⟩
```

If the path ends because the current ray doesn't intersect any objects in the scene, the ray and its throughput α are returned to the caller if the respective pointers are non-NULL.

⟨*Handle ray that leaves the scene during path generation*⟩ ≡ 843
```
if (escapedAlpha) *escapedAlpha = alpha;
if (escapedRay)   *escapedRay = ray;
break;
```

Otherwise, an intersection has been found and the PathVertex::isect member has already been initialized by the call to Scene::Intersect(). The current throughput, the BSDF at the intersection, and the previous direction are now stored in the vertex.

⟨*Record information for current path vertex*⟩ ≡ 843
```
v.alpha = alpha;
BSDF *bsdf = v.isect.GetBSDF(ray, arena);
v.bsdf = bsdf;
v.wPrev = -ray.d;
```

The outgoing direction from the vertex is found using the usual BSDF sampling routines; note that the sample values to initialize the BSDFSample are taken from the appropriate PathSample for the current vertex of the path.

⟨*Sample direction for outgoing Metropolis path direction*⟩ ≡ 843
```
float pdf;
BxDFType flags;
Spectrum f = bsdf->Sample_f(-ray.d, &v.wNext, samples[length].bsdfSample,
                            &pdf, BSDF_ALL, &flags);
v.specularBounce = (flags & BSDF_SPECULAR) != 0;
v.nSpecularComponents = bsdf->NumComponents(BxDFType(BSDF_SPECULAR |
                                BSDF_REFLECTION | BSDF_TRANSMISSION));
if (f.IsBlack() || pdf == 0.f)
    return length+1;
```

Finally, pathScale is computed: it gives the incremental factor that updates the path throughput alpha. The magnitude of pathScale is used in a Russian roulette test: the lower it is, the more likely that the path will be terminated. If the path is not terminated, the throughput is updated appropriately, accounting for both pathScale and the adjustment term that compensates for the Russian roulette test. Before the next pass through the loop, the ray to be traced to find the next path vertex is initialized.

⟨*Terminate path with RR or prepare for finding next vertex*⟩ ≡ 843
```
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Spectrum pathScale = f * AbsDot(v.wNext, n) / pdf;
float rrSurviveProb = min(1.f, pathScale.y());
if (samples[length].rrSample > rrSurviveProb)
    return length+1;
alpha *= pathScale / rrSurviveProb;
ray = RayDifferential(p, v.wNext, ray, v.isect.rayEpsilon);
```

AbsDot() 61
BSDF 478
BSDF::dgShading 479
BSDF::Sample_f() 706
BSDFSample 705
BSDF_ALL 428
BSDF_REFLECTION 428
BSDF_SPECULAR 428
BSDF_TRANSMISSION 428
BxDFType 428
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Intersection::GetBSDF() 484
Intersection::rayEpsilon 186
Normal 65
PathSample 838
PathSample::bsdfSample 838
PathSample::rrSample 838
PathVertex::alpha 843
PathVertex::bsdf 843
PathVertex::isect 843
PathVertex::specularBounce 843
PathVertex::wNext 843
PathVertex::wPrev 843
Point 63
Ray::d 67
RayDifferential 69
Spectrum 263
Spectrum::IsBlack() 265
Spectrum::y() 273

15.7.4 PATH CONTRIBUTIONS

Next, we will define the method that computes the radiance $L(\mathbf{X})$ for a given vector of sample values that represent light-carrying paths of a sequence of points on surfaces in the scene. This method first uses the `GeneratePath()` method to convert the `PathSamples` for the camera and possibly a light source into one or two paths over surfaces in the scene before computing the radiance they transport. The `cameraPath` and `lightPath` parameters should be allocated by the caller to each have length at least equal to `MetropolisRenderer::maxDepth` (though `lightPath` can be `NULL` if bidirectional path tracing isn't being used).

⟨*Metropolis Method Definitions*⟩ +≡

```
Spectrum MetropolisRenderer::PathL(const MLTSample &sample,
        const Scene *scene, MemoryArena &arena, const Camera *camera,
        const Distribution1D *lightDistribution,
        PathVertex *cameraPath, PathVertex *lightPath,
        RNG &rng) const {
    ⟨Generate camera path from camera path samples 845⟩
    if (!bidirectional) {
        ⟨Compute radiance along path using path tracing 846⟩
    }
    else {
        ⟨Sample light ray and apply bidirectional path tracing 846⟩
    }
}
```

Regardless of whether regular path tracing or bidirectional path tracing is being used to compute values of the radiance function, it is necessary to compute a path of intersections with scene surfaces starting from the camera. After the initial ray is generated by the camera, its ray differentials are scaled to account for the actual average number of samples being taken in each pixel; if the samples are equidistantly spaced in both dimensions, their average distance will be one over the square root of the total number of samples taken in the pixel.[11]

⟨*Generate camera path from camera path samples*⟩ ≡ 845

```
RayDifferential cameraRay;
float cameraWt = camera->GenerateRayDifferential(sample.cameraSample,
                                                 &cameraRay);
cameraRay.ScaleDifferentials(1.f / sqrtf(nPixelSamples));
RayDifferential escapedRay;
Spectrum escapedAlpha;
uint32_t cameraLength = GeneratePath(cameraRay, cameraWt, scene, arena,
    sample.cameraPathSamples, cameraPath, &escapedRay, &escapedAlpha);
```

Camera 302
Camera::
GenerateRayDifferential() 303
Distribution1D 648
GeneratePath() 843
MemoryArena 1015
MetropolisRenderer::
maxDepth 853
MLTSample 839
MLTSample::
cameraPathSamples 839
MLTSample::cameraSample 839
PathVertex 843
RayDifferential 69
RayDifferential::
ScaleDifferentials() 70
RNG 1003
Scene 22
Spectrum 263

11 In practice, they aren't equidistantly spaced, since they are generated by random mutations that don't ensure stratification (other than the large step mutations). However, for large numbers of samples, this approximation is usually fine: any aliasing due to insufficient filtering from too-small differentials is generally not noticeable due to the large number of samples taken in the area around each pixel.

If regular path tracing is being used, then the `Lpath()` method (to be defined shortly) computes the radiance carried along the sampled path.

⟨*Compute radiance along path using path tracing*⟩ ≡ 845, 846
```
return Lpath(scene, cameraPath, cameraLength, arena,
             sample.lightingSamples, rng, sample.cameraSample.time,
             lightDistribution, escapedRay, escapedAlpha);
```

For bidirectional path tracing, a light source and a ray leaving the light are sampled. If this sampling step fails to generate a ray that carries any radiance, then radiance along the previously sampled camera path is still computed using regular path tracing. A number of cases may cause the sampled light to return a zero-radiance ray; for example, if the light uses a texture map to describe its emission profile, some regions of the texture are black, and a uniform sampling density is used. In this case, the camera path may still carry radiance, so it's worthwhile to do the standard direct lighting calculation at each path vertex that the path tracing code performs.

⟨*Sample light ray and apply bidirectional path tracing*⟩ ≡ 845
```
⟨Choose light and sample ray to start light path 846⟩
if (lightWt.IsBlack() || lightRayPdf == 0.f) {
    ⟨Compute radiance along path using path tracing 846⟩
}
else {
    ⟨Compute radiance along paths using bidirectional path tracing 846⟩
}
```

⟨*Choose light and sample ray to start light path*⟩ ≡ 846
```
float lightPdf, lightRayPdf;
uint32_t lightNum = lightDistribution->SampleDiscrete(sample.lightNumSample,
                                                      &lightPdf);
const Light *light = scene->lights[lightNum];
Ray lightRay;
Normal Nl;
LightSample lrs(sample.lightRaySamples[0], sample.lightRaySamples[1],
                sample.lightRaySamples[2]);
Spectrum lightWt = light->Sample_L(scene, lrs, sample.lightRaySamples[3],
    sample.lightRaySamples[4], sample.cameraSample.time, &lightRay,
    &Nl, &lightRayPdf);
```

Given the usual case of a light-carrying ray being successfully sampled from the light source, `GeneratePath()` traces the corresponding rays through the scene to compute a path of scattering vertices, using the `lightPathSamples` values for sampling the BSDF at each intersection. Given both the camera and light paths now, `Lbidir()` computes the radiance carried along them.

⟨*Compute radiance along paths using bidirectional path tracing*⟩ ≡ 846
```
lightWt *= AbsDot(Normalize(Nl), lightRay.d) / (lightPdf * lightRayPdf);
uint32_t lightLength = GeneratePath(RayDifferential(lightRay), lightWt,
    scene, arena, sample.lightPathSamples, lightPath, NULL, NULL);
```

AbsDot() 61
CameraSample::time 342
Distribution1D::
SampleDiscrete() 650
GeneratePath() 843
Light 606
Light::Sample_L() 608
LightSample 710
MetropolisRenderer::
Lbidir() 849
MetropolisRenderer::
Lpath() 847
MLTSample::cameraSample 839
MLTSample::
lightingSamples 839
MLTSample::
lightNumSample 839
MLTSample::
lightPathSamples 839
MLTSample::
lightRaySamples 839
Normal 65
Ray 66
Ray::d 67
RayDifferential 69
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265
Vector::Normalize() 63

```
    return Lbidir(scene, cameraPath, cameraLength, lightPath, lightLength,
        arena, sample.lightingSamples, rng, sample.cameraSample.time,
        lightDistribution, escapedRay, escapedAlpha);
```

The `MetropolisRenderer::Lpath()` method is best understood as a refactored implementation of `PathIntegrator::Li()`. Here, the geometric part of constructing the path from the camera by sampling the BSDF at each intersection point has been handled by the `GeneratePath()` method and the path is stored explicitly in an array of `PathVertex` structures. The `Lpath()` method then computes the light carried by the path by sampling the direct illumination at each vertex, scaling it by the path's throughput at the vertex and summing the resulting values. In contrast, the computation to generate the path and the computation to compute direct illumination at each path vertex are comingled in the `PathIntegrator`'s `Li()` method, though essentially the same computation is performed there.

⟨*Metropolis Method Definitions*⟩ +≡

```
Spectrum MetropolisRenderer::Lpath(const Scene *scene,
        const PathVertex *cameraPath, int cameraPathLength,
        MemoryArena &arena, const vector<LightingSample> &samples,
        RNG &rng, float time, const Distribution1D *lightDistribution,
        const RayDifferential &eRay, const Spectrum &eAlpha) const {
    Spectrum L = 0.;
    bool previousSpecular = true, allSpecular = true;
    for (int i = 0; i < cameraPathLength; ++i) {
        ⟨Initialize basic variables for camera path vertex 847⟩
        ⟨Add emitted light from vertex if appropriate 848⟩
        ⟨Compute direct illumination for Metropolis path vertex 848⟩
        L += Ld;
    }
    ⟨Add contribution of escaped ray, if any 849⟩
    return L;
}
```

BSDF::dgShading 479
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DirectLightingIntegrator 742
Distribution1D 648
LightingSample 838
MemoryArena 1015
MetropolisRenderer 852
Normal 65
PathIntegrator::Li() 767
PathVertex 843
PathVertex::bsdf 843
Point 63
RayDifferential 69
RNG 1003
Scene 22
Spectrum 263

⟨*Initialize basic variables for camera path vertex*⟩ ≡ 847, 849

```
const PathVertex &vc = cameraPath[i];
const Point &pc = vc.bsdf->dgShading.p;
const Normal &nc = vc.bsdf->dgShading.nn;
```

The `MetropolisRenderer` can be configured to compute direct illumination using the standard approach as implemented by the `DirectLightingIntegrator`, using Metropolis sampling only for indirect illumination. For scenes where the direct lighting is handled well with the usual algorithms, doing so allows the `MetropolisRenderer` to concentrate its sampling effort on the more difficult portions of the problem. However, this option makes the implementations of `Lpath()` and `Lbidir()` more complex than they would be otherwise: with the lighting integral possibly partitioned, it is necessary to be careful to not double-count any potential light-carrying paths and to not miss any potential light-carrying paths.

If direct lighting is handled separately, then the `directLighting` member variable is non-`NULL`. The separate direct lighting computation includes light emitted at the first intersection point seen by the camera, direct illumination at that point, and emission and direct illumination at any surfaces intersected by sequences of perfect specular reflection or transmission events starting from the first intersection point. Given all this, we can determine whether emission at the current path vertex should be included in the radiance value returned by `Lpath()` as follows:

- If direct lighting is being handled separately, then we only include emission if scattering at the last vertex was specular reflection or transmission and if all of the path's vertices haven't been specular. If the last vertex was nonspecular, then we must ignore emission from this surface, since the direct lighting calculation performed as part of path tracing at the previous vertex is responsible for accounting for this portion of the light transport. If all of the vertices in the path up to this one were specular, then the emission here will have been included by the direct lighting integrator and so should also be ignored here.
- If Metropolis sampling is also responsible for computing direct lighting, then emission is included only if the previous scattering event was specular: in the nonspecular case, path tracing's direct lighting calculation at the previous vertex accounts for illumination scattered by nonspecular components of the BSDF.

⟨*Add emitted light from vertex if appropriate*⟩ ≡ **847, 850**
```
if (previousSpecular && (directLighting == NULL || !allSpecular))
    L += vc.alpha * vc.isect.Le(vc.wPrev);
```

Following similar logic, `Lpath()` skips performing a direct lighting computation at the vertex in cases where a separate direct lighting computation was performed and where that computation has accounted for the current type of light transport path.

⟨*Compute direct illumination for Metropolis path vertex*⟩ ≡ **847, 850**
```
Spectrum Ld(0.f);
if (directLighting == NULL || !allSpecular) {
    ⟨Choose light and call EstimateDirect() for Metropolis vertex 848⟩
}
previousSpecular = vc.specularBounce;
allSpecular &= previousSpecular;
```

Reflection due to direct lighting at this vertex is computed using the regular direct lighting routines. A single light is sampled according to the lights' relative power, and a sample is taken from it using the sample values in the appropriate `LightingSample`. The `alpha` component of the current `PathVertex` provides the scaling factor that accounts for scattering by the BSDFs at the previous vertices in the path.

⟨*Choose light and call* `EstimateDirect()` *for Metropolis vertex*⟩ ≡ **848**
```
const LightingSample &ls = samples[i];
float lightPdf;
uint32_t lightNum = lightDistribution->SampleDiscrete(ls.lightNum,
                                                      &lightPdf);
const Light *light = scene->lights[lightNum];
```

BSDF_ALL 428
BSDF_SPECULAR 428
BxDFType 428
Distribution1D::
SampleDiscrete() 650
EstimateDirect() 749
Intersection::Le() 625
Intersection::rayEpsilon 186
Light 606
LightingSample 838
LightingSample::
bsdfSample 838
LightingSample::lightNum 838
LightingSample::
lightSample 838
MetropolisRenderer::
directLighting 853
PathVertex 843
PathVertex::alpha 843
PathVertex::isect 843
PathVertex::
specularBounce 843
PathVertex::wPrev 843
Scene::lights 23
Spectrum 263

```
Ld = vc.alpha *
     EstimateDirect(scene, this, arena, light, pc, nc, vc.wPrev,
                    vc.isect.rayEpsilon, time, vc.bsdf, rng,
                    ls.lightSample, ls.bsdfSample,
                    BxDFType(BSDF_ALL & ~BSDF_SPECULAR)) / lightPdf;
```

Finally, if the camera path ended because one of the rays in the path didn't intersect any primitives and left the scene, it may be necessary to add the contribution of emitted light from infinite light sources. As before, this computation is only performed if this illumination wasn't accounted for by a separate direct lighting computation or the direct lighting computation performed at the previous vertex.

⟨*Add contribution of escaped ray, if any*⟩ ≡ 847, 849

```
if (!eAlpha.IsBlack() && previousSpecular &&
    (directLighting == NULL || !allSpecular))
    for (uint32_t i = 0; i < scene->lights.size(); ++i)
       L += eAlpha * scene->lights[i]->Le(eRay);
```

`Lbidir()` uses bidirectional path tracing to compute the radiance carried by a camera path and a light path. Recall the discussion from Section 15.3.5 of how the estimates of the path integral form of the light transport equation can be evaluated with a bidirectional approach: for each prefix of the camera path and the light path, a shadow ray is traced between the corresponding two path vertices to construct a light-carrying path. If the points are mutually visible, the full path's contribution is added to the estimate.

⟨*Metropolis Method Definitions*⟩ +≡

```
Spectrum MetropolisRenderer::Lbidir(const Scene *scene,
        const PathVertex *cameraPath, int cameraPathLength,
        const PathVertex *lightPath, int lightPathLength,
        MemoryArena &arena, const vector<LightingSample> &samples,
        RNG &rng, float time, const Distribution1D *lightDistribution,
        const RayDifferential &eRay, const Spectrum &eAlpha) const {
    Spectrum L = 0.;
    bool previousSpecular = true, allSpecular = true;
    ⟨Compute number of specular vertices for each path length 850⟩
    for (int i = 0; i < cameraPathLength; ++i) {
        ⟨Initialize basic variables for camera path vertex 847⟩
        ⟨Compute reflected light at camera path vertex 850⟩
    }
    ⟨Add contribution of escaped ray, if any 849⟩
    return L;
}
```

Distribution1D 648
Light::Le() 631
LightingSample 838
MemoryArena 1015
MetropolisRenderer::
directLighting 853
PathVertex 843
RayDifferential 69
RNG 1003
Scene 22
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265

When connecting camera paths and light paths to generate a path of particular length in the following, it will be necessary to compute how many other paths of the same length will be created by other bidirectional connections (for example, by using one fewer camera path vertex and one more light path vertex, and so forth). Path vertices with specular reflection or transmission introduce a complication to this computation: if one of the two respective BSDFs at vertices being connected is specular, then we don't bother

tracing a ray to generate a path, since the value of the BSDF is guaranteed to be zero. However, this means that there is one fewer way of generating a path of the corresponding length than there would have been otherwise—this must be accounted for when adding up all of the different ways of generating paths of this length.

To help with this computation, here we initialize an array, `nSpecularVertices`, such that the ith element records how many of the paths with i vertices that can be generated from this camera and light path have specular BSDFs. Note that the first two entries of this array are always zero; while this could be avoided by allocating a smaller array and changing its indexing below, we prefer a simpler indexing scheme that can be easily understood when used in code below.

⟨*Compute number of specular vertices for each path length*⟩ ≡ 849

```
int nVerts = cameraPathLength + lightPathLength + 2;
int *nSpecularVertices = ALLOCA(int, nVerts);
memset(nSpecularVertices, 0, nVerts * sizeof(int));
for (int i = 0; i < cameraPathLength; ++i)
    for (int j = 0; j < lightPathLength; ++j)
        if (cameraPath[i].specularBounce || lightPath[j].specularBounce)
            ++nSpecularVertices[i+j+2];
```

Emission and direct lighting at each camera vertex are computed in the same manner as in the `Lpath()` method, reusing the same code fragments. There are two important details here. First, the direct lighting contribution at this vertex computed by the ⟨*Compute direct illumination for Metropolis path vertex*⟩ fragment, `Ld`, is scaled by the reciprocal of the number of vertices in the path $(i + 1)$ minus the number of paths of this length that end with specular vertices. This factor accounts for all of the different ways that bidirectional path tracing may generate a light-carrying path of the same length.

To understand this factor, consider the direct illumination calculation at the second vertex, $i = 1$, of the camera path: direct lighting at this vertex corresponds to illumination that has bounced off of two surfaces in the scene between the camera and the light. However, bidirectional path tracing can also construct a two-bounce path by connecting the first vertex in the camera path with the first vertex in the light path with a shadow ray. There are thus two ways of constructing two bounce paths, so these two paths must be reweighted with weights that sum to one so that they don't double-count this mode of light transport; a straightforward choice is to weight each of them by 1/2. However, if either or both of the first vertices in the camera or light path was specular, then the corresponding path would have no contribution, and the appropriate weight would be $1/(2 - 1) = 1$. In general, in the absence of specular reflection, there are n ways for bidirectional path tracing to construct an n-bounce light-carrying path.

⟨*Compute reflected light at camera path vertex*⟩ ≡ 849

```
⟨Add emitted light from vertex if appropriate 848⟩
⟨Compute direct illumination for Metropolis path vertex 848⟩
L += Ld / (i + 1 - nSpecularVertices[i+1]);
if (!vc.specularBounce) {
    ⟨Loop over light path vertices and connect to camera vertex 851⟩
}
```

ALLOCA() 1009

PathVertex::
specularBounce 843

If the current camera path vertex is nonspecular, then the corresponding paths connecting it to each of the light path vertices are created.

⟨*Loop over light path vertices and connect to camera vertex*⟩ ≡ 850
```
for (int j = 0; j < lightPathLength; ++j) {
    const PathVertex &vl = lightPath[j];
    const Point &pl = vl.bsdf->dgShading.p;
    const Normal &nl = vl.bsdf->dgShading.nn;
    if (!vl.specularBounce) {
        ⟨Compute contribution between camera and light vertices 851⟩
    }
}
```

Given a particular path, we must determine how to compute the overall contribution of the path we've constructed. Specifically, we are evaluating a particular term of the path integral form of the light transport equation, Equation (15.5), where the path length n is the sum of the number of camera and light vertices in the current path. If we consider the path contribution Equation (15.6), we have constructed camera and light subpaths with the incremental path construction approach used in Section 15.3.3 for regular path tracing. Given the throughput of these paths up to the current vertices, $T(\bar{\mathrm{p}}_i)$ and $T(\bar{\mathrm{q}}_j)$, respectively, we can find that the contribution of a path of i camera vertices and j light vertices is given by

$$P(\bar{\mathrm{p}}_{i,j}) = W_{\mathrm{e}}\, T(\bar{\mathrm{p}}_i) \left[f(\mathrm{p}_{i-1} \to \mathrm{p}_i \to \mathrm{q}_j) G(\mathrm{p}_i \leftrightarrow \mathrm{q}_j) f(\mathrm{p}_i \to \mathrm{q}_j \to \mathrm{q}_{j-1}) \right] T(\bar{\mathrm{q}}_j)\, L_{\mathrm{e}}.$$

The values of two throughput-based terms, $W_{\mathrm{e}} T(\bar{\mathrm{p}}_i)$ for the camera path and $T(\bar{\mathrm{q}}_j) L_{\mathrm{e}}$ for the light path, are already available in `cameraPath[i].alpha` and `lightPath[j].alpha`, respectively, so we only need to compute the term in brackets to find the path's overall contribution. The various values needed to do so are available in the two `PathVertex` structures. One additional detail related to specular reflection is that when evaluating a BSDF that also has specular components an additional scaling factor based on the number of specular components is used to scale the nonspecular BSDF value; this accounts for the PDF for sampling this component of the complete BSDF.

⟨*Compute contribution between camera and light vertices*⟩ ≡ 851
```
Vector w = Normalize(pl - pc);
Spectrum fc = vc.bsdf->f(vc.wPrev, w) * (1 + vc.nSpecularComponents);
Spectrum fl = vl.bsdf->f(-w, vl.wPrev) * (1 + vl.nSpecularComponents);
if (fc.IsBlack() || fl.IsBlack()) continue;
Ray r(pc, pl - pc, 1e-3f, .999f, time);
if (!scene->IntersectP(r)) {
    ⟨Compute weight for bidirectional path, pathWt 852⟩
    float G = AbsDot(nc, w) * AbsDot(nl, w) / DistanceSquared(pl, pc);
    L += (vc.alpha * fc * G * fl * vl.alpha) * pathWt;
}
```

As described above, bidirectional path tracing will construct n separate instances of paths with n vertices. In the aggregate, all of the paths of a particular length must be weighted so that collectively the sum of their weights is one so they make the correct contribution.

AbsDot() 61
BSDF::dgShading 479
BSDF::f() 481
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
DistanceSquared() 65
Normal 65
PathVertex 843
PathVertex::alpha 843
PathVertex::bsdf 843
PathVertex::nSpecularComponents 843
PathVertex::specularBounce 843
PathVertex::wPrev 843
Point 63
Ray 66
Scene::IntersectP() 24
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
Vector::Normalize() 63

In the implementation here, each path is weighted equally. A more effective approach is to weight them using multiple importance sampling, but this improvement is left for Exercise 15.26.

There are two things to note in this computation: first, 2 is added to the sum of i and j to get the number of path vertices. Because arrays are indexed from zero, doing so gives the actual number of vertices in the path. Second, the number of specular vertices in paths of the same length as this one is subtracted from the weight to account for the fact that we don't connect the two paths when we encounter specular vertices, so we are actually making fewer than n connections for paths with n vertices in that case.

⟨*Compute weight for bidirectional path,* `pathWt`⟩ ≡ 851
```
float pathWt = 1.f / (i + j + 2 - nSpecularVertices[i+j+2]);
```

15.7.5 `MetropolisRenderer` IMPLEMENTATION

Given all of this infrastructure—an explicit representation of an n-dimensional sample $\mathbf{X}$, functions to apply mutations to it, and the ability to evaluate the radiance and scalar contribution functions for a given sample value—we can move forward to the heart of the implementation of the `MetropolisRenderer`.

⟨*Metropolis Declarations*⟩ ≡
```
class MetropolisRenderer : public Renderer {
public:
    ⟨MetropolisRenderer Public Methods⟩
private:
    ⟨MetropolisRenderer Private Methods⟩
    ⟨MetropolisRenderer Private Data 853⟩
};
```

The `MetropolisRenderer` constructor, not shown here, initializes the following member variables from parameters provided to it. Most of the member variables are self-explanitory: `camera` holds the `Camera` specified in the scene description file, `bidirectional` is `true` if bidirectional path tracing is to be used to evaluate sample contributions, and if the direct lighting calculation is being done separately from Metropolis sampling then `directLighting` is non-`NULL` and `nDirectPixelSamples` holds the number of samples to be taken per pixel for direct lighting.

The maximum length of a camera path (and light path, if bidirectional path tracing is used) is given by `maxDepth`, and the average number of samples per pixel to take is given by `nPixelSamples`. (In other words, the total number of Metropolis samples taken will be the product of the number of pixels and `nPixelSamples`, but pixels will receive an actual number of samples related to their brightness, due to Metropolis's property of taking more samples in regions where the function's value is high.)

The number of large step mutations for each pixel is given by `largeStepsPerPixel`. Thus, the overall probability of doing a large step mutation rather than a small step mutation is `largeStepsPerPixel / nPixelSamples`.

Camera 302
Renderer 24

Finally, nBootstrap sets the number of "bootstrapping" samples, where the integral of the scalar contribution function is computed, and maxConsecutiveRejects limits the number of successive rejections of proposed mutations; if a large number of mutations are rejected in a row, we override the sample acceptance computation and force acceptance. A long series of rejected mutations should only happen very rarely, but, when it does, image results are improved if we force the sampler to move on, visiting other pixels in the image. This parameter does introduce a small amount of bias in the final result; it should be set to a large value if unbiased rendering is absolutely required.

⟨*MetropolisRenderer Private Data*⟩ ≡ 852

```
Camera *camera;
bool bidirectional;
uint32_t nDirectPixelSamples, nPixelSamples, maxDepth;
uint32_t largeStepsPerPixel, nBootstrap, maxConsecutiveRejects;
DirectLightingIntegrator *directLighting;
```

The Render() method has four main responsibilities. First, if direct lighting is being calculated with the DirectLightingIntegrator, it launches tasks to do this computation. Next, it computes the integral of the scalar contribution function, b. (Recall the need for this value from Equation (15.21).) In order to start Metropolis sampling, a suitable initial sample must be found; this task is handled by the ⟨*Select initial sample from bootstrap samples*⟩ fragment. Finally, this method launches tasks to perform Metropolis sampling before writing out the final rendered image.

There are a few trade-offs to consider in deciding whether to use conventional techniques for the direct lighting lighting calculation. One advantage that the approach implemented in the DirectLightingIntegrator has is that the samples over the light sources can easily be well distributed with low-discrepancy point sets; doing so can substantially improve efficiency. MLT's property of generating samples that are distributed according to the PDF of the function being sampled can also be a disadvantage: if direct lighting makes a substantial contribution to some or all of the image's overall brightness, then MLT will focus the majority of its sampling effort on resolving direct lighting, rather than the potentially more difficult indirect lighting paths. More generally, if there are portions of the radiance function that both make a relatively large contribution and can be handled efficiently with conventional techniques, overall efficiency can be improved if we partition the radiance function so that we can handle the easy parts with conventional approaches and leave the difficult bits for MLT to focus its effort on.

Figure 15.31 compares rendering the San Miguel scene with MLT with direct lighting included and not included, for equal amounts of computation. For this scene, direct lighting is well handled by conventional techniques, so it's more efficient to not include it in the Metropolis sampling phase of rendering.

However, always computing direct lighting separately is not always the right approach. If direct lighting can't be sampled efficiently by standard techniques, then MLT may be a better choice for it. For example, if the majority of the light sources are completely occluded, or if the scene is lit by an infinite area light source with a very bright region

Camera 302
DirectLightingIntegrator 742



(a)



(b)

Figure 15.31: Comparison of Rendering the San Miguel Scene with Direct Lighting Included in Metropolis Sampling and Not Included. (a) Rendered with MLT with 1024 samples per pixel, with direct lighting included in the Metropolis sampling. (b) Rendered with just 900 Metropolis samples per pixel and 128 direct lighting samples per pixel (approximately equal computation time). For this scene, the direct lighting component is well handled by conventional techniques, so it's more efficient to partition the integrand and handle direct lighting separately, allowing the Metropolis sampling phase to concentrate its sampling work on the indirect illumination and not spend its effort on the relatively easy direct lighting. Note that the shadows under the tables are less noisy when direct lighting is done separately, for example. *(Model courtesy of Guillermo M. Leal Llaguno.)*

that happens to be fully occluded, conventional direct lighting methods will take many samples with zero contribution and Metropolis may be a more efficient choice.

Thus, we leave the decision of whether to separately compute direct lighting up to the user. The fragment ⟨*Compute direct lighting before Metropolis light transport*⟩, not included in the text here, launches tasks to compute direct lighting separately if requested. Its structure is quite similar to that of the main `SamplerRenderer` rendering loop in Section 1.3.4.

⟨*Metropolis Method Definitions*⟩ +≡
```
void MetropolisRenderer::Render(const Scene *scene) {
    if (scene->lights.size() > 0) {
        int x0, x1, y0, y1;
        camera->film->GetPixelExtent(&x0, &x1, &y0, &y1);
        float t0 = camera->shutterOpen, t1 = camera->shutterClose;
        Distribution1D *lightDistribution = ComputeLightSamplingCDF(scene);

        if (directLighting != NULL) {
            ⟨Compute direct lighting before Metropolis light transport⟩
        }
        ⟨Take initial set of samples to compute b 855⟩
        ⟨Select initial sample from bootstrap samples 857⟩
        ⟨Launch tasks to generate Metropolis samples⟩
    }
    camera->film->WriteImage();
}
```

Camera::film 302
Camera::shutterClose 302
Camera::shutterOpen 302
ComputeLightSamplingCDF() 709
Distribution1D 648
Film::GetPixelExtent() 404
Film::WriteImage() 404
MemoryArena 1015
MetropolisRenderer::camera 853
MetropolisRenderer::directLighting 853
MetropolisRenderer::maxDepth 853
MetropolisRenderer::nBootstrap 853
MLTSample 839
PathVertex 843
RNG 1003
SamplerRenderer 25
Scene 22
Scene::lights 23

In order to compute the integral of the scalar contribution function $b = \int I(\mathrm{X})\mathrm{d}\Omega$, we can use the sample generation and evaluation routines implemented previously: the `LargeStep()` function is called to generate uniform random samples over the sample space domain, and `PathL()` computes the value of the function for each one. The `sumI` variable accumulates the sum of the scalar contribution functions. This fragment also stores the contribution of each sample in the `bootstrapI` array; these will be used in a second pass through the samples, below, to select the initial sample for rendering.

⟨*Take initial set of samples to compute b*⟩ ≡ 855
```
RNG rng(0);
MemoryArena arena;
vector<float> bootstrapI;
vector<PathVertex> cameraPath(maxDepth, PathVertex());
vector<PathVertex> lightPath(maxDepth, PathVertex());
float sumI = 0.f;
MLTSample sample(maxDepth);
for (uint32_t i = 0; i < nBootstrap; ++i) {
    ⟨Generate random sample and path radiance for MLT bootstrapping 856⟩
    ⟨Compute contribution for random sample for MLT bootstrapping 856⟩
}
float b = sumI / nBootstrap;
```

Recall that `LargeStep()` expects the caller to pass in uniformly sampled x and y pixel coordinates. Here, we don't worry about stratifying those values over the entire image and just compute uniform random values over the pixel range.

⟨*Generate random sample and path radiance for MLT bootstrapping*⟩ ≡ 855
```
float x = Lerp(rng.RandomFloat(), x0, x1);
float y = Lerp(rng.RandomFloat(), y0, y1);
LargeStep(rng, &sample, maxDepth, x, y, t0, t1, bidirectional);
Spectrum L = PathL(sample, scene, arena, camera, lightDistribution,
                   &cameraPath[0], &lightPath[0], rng);
```

After the path's radiance value has been computed, the `I()` function computes its scalar contribution.

⟨*Compute contribution for random sample for MLT bootstrapping*⟩ ≡ 855
```
float I = ::I(L);
sumI += I;
bootstrapI.push_back(I);
arena.FreeAll();
```

As described at the start of the section, the spectral radiance value $L(\mathbf{X})$ must be converted to a value given by the scalar contribution function so that the acceptance probability can be computed for the Metropolis sampling algorithm. There is a fair amount of flexibility in how this function can be defined, as long as it is greater than zero whenever the radiance function is nonzero. For a scene where we wanted to apply extra sampling work to caustics, for example, we could artifically increase its value for any light-carrying path that intersected specular surfaces. Or, if a scene had an extremely bright region that was using too many samples, the function's value could be clamped if it was larger than a fixed limit, so that the Metropolis sampler didn't devote too many samples to the very bright part of the scene. (See the "Further Reading" section for more information on these topics.) Here, we just compute the path's luminance, which is a reasonable baseline approach.

⟨*Metropolis Method Definitions*⟩ +≡
```
inline float I(const Spectrum &L) {
    return L.y();
}
```

Following the approach described in the discussion of start-up bias in Section 13.4.3, here we select an initial sample with probability proportional to its contribution relative to the sum of all of the sample contributions computed above. We can do this without explicitly computing a normalized CDF by selecting a uniform random value between zero and the contribution sum and then looping over all of the path samples from the start until we find the one whose contribution causes the accumulated sum of contributions to pass this value. We don't need to save all of the sample values generated during the first pass; instead we re-seed the random number generator to the same seed as before and then apply the same mutations to the `MLTSample`; the resulting sample will then be the appropriate one.

I() 856
LargeStep() 840
Lerp() 1000
MemoryArena::FreeAll() 1017
MetropolisRenderer::bidirectional 853
MetropolisRenderer::camera 853
MetropolisRenderer::maxDepth 853
MetropolisRenderer::PathL() 845
RNG::RandomFloat() 1003
Spectrum 263
Spectrum::y() 273

⟨*Select initial sample from bootstrap samples*⟩ ≡ 855
```
float contribOffset = rng.RandomFloat() * sumI;
rng.Seed(0);
sumI = 0.f;
MLTSample initialSample(maxDepth);
for (uint32_t i = 0; i < nBootstrap; ++i) {
    float x = Lerp(rng.RandomFloat(), x0, x1);
    float y = Lerp(rng.RandomFloat(), y0, y1);
    LargeStep(rng, &initialSample, maxDepth, x, y, t0, t1,
              bidirectional);
    sumI += bootstrapI[i];
    if (sumI > contribOffset)
        break;
}
```

15.7.6 RENDERING

`MLTTask` provides the `Task` implementation used for the parallel execution of MLT in `pbrt`. Rendering is decomposed so that each task is responsible for performing exactly one large step mutation for each pixel in the image (as well as the appropriate number of small step mutations given the parameter settings). For example, if the overall image resolution is 1024×1024 pixels and the user has set the large step probability to be $1/32$ and the average number of pixel samples to be 512, then $512/32 = 16$ `MLTTasks` will be launched, each one taking a total of $(1024 \times 1024) \times 32$ samples, with $1/32$ of them large steps and $31/32$ of them small steps.

By ensuring that each task performs a number of large step mutations exactly equal to the number of pixels, it is possible to ensure that the (x, y) pixel locations of the proposed large step samples are well distributed across the image plane. Specifically, we make sure that in each task each pixel in the image has exactly one proposed large step mutation. Doing so ensures that all pixels receive at least some samples, which otherwise isn't guaranteed by straightforward application of the basic Metropolis sampling algorithm.

⟨*Metropolis Method Definitions*⟩ +≡
```
void MLTTask::Run() {
    ⟨Declare basic MLTTask variables and prepare for sampling 858⟩
    for (uint64_t s = 0; s < nTaskSamples; ++s) {
        ⟨Compute proposed mutation to current sample 859⟩
        ⟨Compute contribution of proposed sample 860⟩
        ⟨Compute acceptance probability for proposed sample 860⟩
        ⟨Splat current and proposed samples to Film 860⟩
        ⟨Randomly accept proposed path mutation (or not) 861⟩
    }
}
```

LargeStep() 840
Lerp() 1000
MetropolisRenderer::
bidirectional 853
MetropolisRenderer::
maxDepth 853
MetropolisRenderer::
nBootstrap 853
MLTSample 839
RNG::RandomFloat() 1003
RNG::Seed() 1003
Task 1041

A number of values derived from the parameter settings are computed before sampling begins; most of these should be self-explanatory.

⟨*Declare basic* MLTTask *variables and prepare for sampling*⟩ ≡ 857
```
uint32_t nPixels = (x1-x0) * (y1-y0);
uint32_t nPixelSamples = renderer->nPixelSamples;
uint32_t largeStepRate = nPixelSamples / renderer->largeStepsPerPixel;
uint64_t nTaskSamples = uint64_t(nPixels) * uint64_t(largeStepRate);
uint32_t consecutiveRejects = 0;
⟨Declare variables for storing and computing MLT samples 858⟩
⟨Compute L[current] for initial sample 858⟩
⟨Compute randomly permuted table of pixel indices for large steps 859⟩
```

A few variables are also needed for computing MLT sample values and storing the current state. First, memory for PathVertex arrays must be preallocated here so that it can be passed in to calls to the PathL() method below. Storage is also allocated for two MLTSamples, spectral radiance values, and scalar contribution function values. These arrays are indexed with the current and proposed variables; in the implementation below, samples[current] will hold the sample values for the current sample, samples[proposed] will hold the sample values for the proposed mutation, and so forth. When proposed mutations are accepted, the values of current and proposed are swapped. This approach allows us to avoid copying substantial amounts of data when a proposed mutation is accepted, as would be necessary if we instead had distinctly named vector<MLTSample> variables for the proposed and current sample.

⟨*Declare variables for storing and computing MLT samples*⟩ ≡ 858
```
MemoryArena arena;
RNG rng(taskNum);
vector<PathVertex> cameraPath(renderer->maxDepth, PathVertex());
vector<PathVertex> lightPath(renderer->maxDepth, PathVertex());
vector<MLTSample> samples(2, MLTSample(renderer->maxDepth));
Spectrum L[2];
float I[2];
uint32_t current = 0, proposed = 1;
```

The initial sample for all tasks is the one found by the bootstrapping process in Section 15.7.5. All MLTTasks redundantly compute its radiance value and sample contribution, though the cost of this extra work relative to the overall amount of computation the tasks perform is tiny. Because each task uses a unique seed to the pseudo-random number generator, each one will quickly mutate away from the same initial sample to a distinct chain of samples through the state space.

⟨*Compute* L[current] *for initial sample*⟩ ≡ 858
```
samples[current] = initialSample;
L[current] = renderer->PathL(initialSample, scene, arena, camera,
                 lightDistribution, &cameraPath[0], &lightPath[0], rng);
I[current] = ::I(L[current]);
arena.FreeAll();
```

Each pixel in the image should receive a single proposed large step mutation for each task. So that the transition densities between samples for large step mutations is uniform, the order in which pixels are visited by the large step mutations should be randomized.

I() 856
MemoryArena 1015
MemoryArena::FreeAll() 1017
MetropolisRenderer::
 largeStepsPerPixel 853
MetropolisRenderer::
 maxDepth 853
MetropolisRenderer::
 nPixelSamples 853
MetropolisRenderer::
 PathL() 845
MLTSample 839
PathVertex 843
RNG 1003
Spectrum 263

(It should also be different in different tasks.) The bookkeeping for this is handled by assigning each pixel an index and randomly shuffling an array of the pixel indices. The code that calls `LargeStep()` below will in turn convert pixel indices back to (x, y) pixel values.

⟨*Compute randomly permuted table of pixel indices for large steps*⟩ ≡ 858

```
uint32_t pixelNumOffset = 0;
vector<int> largeStepPixelNum;
for (uint32_t i = 0; i < nPixels; ++i) largeStepPixelNum.push_back(i);
Shuffle(&largeStepPixelNum[0], nPixels, 1, rng);
```

The implementation of the Metropolis sampling routine in the following fragments follows the pseudo-code that uses the expected values technique from Section 13.4.1: a mutation is proposed, the value of the function for the mutated sample and the acceptance probability are computed, the contributions of both the new and old samples are recorded, and the proposed mutatation is randomly accepted based on its acceptance probability.

The implementation here regularly alternates between the large step and the small step mutations, taking one large step and then `largeStepRate-1` small steps, another large step, and so forth. Although it could instead randomly choose between the two mutations with corresponding probabilities, with the approach implemented here we ensure that this task will perform exactly one large step mutation for each pixel.

When a large step mutation does occur, the (x, y) image location to pass to the large step is found in a two-step process. First, the current index in the shuffled pixel index array is converted to an integer (x, y) pixel coordinate. Next, an offset in $[0, 1]^2$ is added to the integer pixel coordinate. The same offset is used for *all* of the large steps in this task; the offset values are computed by the code that launches the task and stored in the `MLTTask::dx`, `MLTTask::dy` member variables. To compute these values, the code that launches `MLTTasks` computes a low-discrepancy sampling pattern in $[0, 1]^2$, with one sample for each task. Thus, in the aggregate when all of the tasks have finished, not only will each pixel have received the same number of proposed large step mutations, but also their (x, y) pixel locations will be well distributed within each pixel's area.

⟨*Compute proposed mutation to current sample*⟩ ≡ 857

```
samples[proposed] = samples[current];
bool largeStep = ((s % largeStepRate) == 0);
if (largeStep) {
    int x = x0 + largeStepPixelNum[pixelNumOffset] % (x1 - x0);
    int y = y0 + largeStepPixelNum[pixelNumOffset] / (x1 - x0);
    LargeStep(rng, &samples[proposed], renderer->maxDepth,
              x + dx, y + dy, t0, t1, renderer->bidirectional);
    ++pixelNumOffset;
}
else
    SmallStep(rng, &samples[proposed], renderer->maxDepth,
              x0, x1, y0, y1, t0, t1, renderer->bidirectional);
```

LargeStep() 840
MetropolisRenderer::bidirectional 853
MetropolisRenderer::maxDepth 853
Shuffle() 354
SmallStep() 842

Given the proposed sample, the previously defined `PathL()` and `I()` methods compute the sample's radiance value and scalar contribution.

⟨*Compute contribution of proposed sample*⟩ ≡ 857
```
L[proposed] = renderer->PathL(samples[proposed], scene, arena, camera,
                lightDistribution, &cameraPath[0], &lightPath[0], rng);
I[proposed] = ::I(L[proposed]);
arena.FreeAll();
```

Since all of the mutations here are symmetric, the transition probabilities cancel when computing the acceptance probability and so the acceptance probability for the proposed sample is found using Equation (13.8).

⟨*Compute acceptance probability for proposed sample*⟩ ≡ 857
```
float a = min(1.f, I[proposed] / I[current]);
```

The contributions for the proposed and current samples are computed using Equation (15.21) and the two samples are splatted into the image using the `Film::Splat()` method. Before being provided to the film, they are scaled with weights based on the expected values optimization introduced in Section 13.4.1.[12] Note that `Film::Splat()` is responsible for incorporating the contribution of filter function $h_j(\mathbf{X})$ before storing the sample's contribution in the pixels that it contributes to.

⟨*Splat current and proposed samples to* `Film`⟩ ≡ 857
```
if (I[current] > 0.f) {
    Spectrum contrib =  (b / nPixelSamples) * L[current] / I[current];
    camera->film->Splat(samples[current].cameraSample,
                        (1.f - a) * contrib);
}
if (I[proposed] > 0.f) {
    Spectrum contrib =  (b / nPixelSamples) * L[proposed] / I[proposed];
    camera->film->Splat(samples[proposed].cameraSample,
                        a * contrib);
}
```

Finally, the proposed mutation is either accepted or rejected, based on the computed acceptance probability `a` (and the fixed limit of successive rejections that are allowed). If the mutation is accepted, then the values of the `current` and `proposed` indices are exchanged; because these variables can have only have the values zero or one, then taking the exclusive-OR of each of them with one does this efficiently.

12 Kelemen et al. (2002) suggested a slightly more efficient implementation of this logic where only the sample that is rejected is splatted to the film, and the contribution of the accepted sample is accumulated in a local variable here until it is eventually rejected. This approach cuts in half the number of `Film::Splat()` calls while still computing the same result.

Camera::film 302
Film::Splat() 403
I() 856
MemoryArena::FreeAll() 1017
MetropolisRenderer::PathL() 845
MLTSample::cameraSample 839
Spectrum 263

⟨*Randomly accept proposed path mutation (or not)*⟩ ≡ 857

```
if (consecutiveRejects >= renderer->maxConsecutiveRejects ||
    rng.RandomFloat() < a) {
    current ^= 1;
    proposed ^= 1;
    consecutiveRejects = 0;
}
else
    ++consecutiveRejects;
```

FURTHER READING

The first application of Monte Carlo to global illumination for creating synthetic images that we are aware of was described in Tregenza's paper on lighting design (Tregenza 1983). Cook's distribution ray-tracing algorithm computed glossy reflections, soft shadows from area lights, motion blur, and depth of field with Monte Carlo sampling (Cook, Porter, and Carpenter 1984; Cook 1986), although the general form of the light transport equation wasn't stated until papers by Kajiya (1986) and Immel, Cohen, and Greenberg (1986). Kajiya's paper also introduced the path-tracing algorithm for solving the light transport equation.

Additional important theoretical work on light transport has been done by Arvo (1993, 1995), who has investigated the connection between rendering algorithms in graphics and previous work in *transport theory*, which applies classical physics to particles and their interactions to predict their overall behavior. Our description of the path integral form of the LTE follows the framework in Veach's Ph.D. thesis, which has thorough coverage of different forms of the LTE and its mathematical structure (Veach 1997). Christensen (2003) wrote a paper on the application of adjoint functions and importance to solving the LTE; it gives a comprehensive overview of related topics with many pointers to further information.

Russian roulette was introduced to graphics by Arvo and Kirk (1990). Hall and Greenberg (1983) had previously suggested adaptively terminating ray trees by not tracing rays with less than some minimum contribution. Arvo and Kirk's technique is unbiased, although in some situations, bias and less noise may be the less undesirable artifact.

A number of approaches have been developed to efficiently render scenes with hundreds or thousands of light sources. (For densely occluded environments, many of the lights in the scene may have little or no contribution to the part of the scene visible from the camera.) Early work on this issue was done by Ward (1991a) and Shirley et al. (1996). Wald et al. (2003) suggested rendering an image with path tracing and a very low sampling rate (e.g., one path per pixel), recording information about which of the light sources made some contribution to the image. This information is then used to set probabilities for sampling each light. Donikian et al. (2006) adaptively found PDFs for sampling lights through an iterative process of taking a number of light samples, noting which ones were effective, and reusing this information at nearby pixels. The "lightcuts" approach, described in a few paragraphs, also addresses this problem.

MetropolisRenderer::
maxConsecutiveRejects 853

RNG::RandomFloat() 1003

pbrt's direct lighting routines are based on using multiple importance sampling to combine samples taken from the BSDF and the light sources' sampling distributions; while this works well in many cases, it can be ineffective in cases where the product of these two functions has a significantly different distribution than either one individually. A number of more efficient approaches have been developed to sample directly from the product distribution (Burke et al. 2005; Ghosh et al. 2005; Cline et al. 2006). Clarberg, Rousselle, and collaborators developed techniques based on representing BSDFs and illumination in the wavelet basis and efficiently sampling from their product (Clarberg et al. 2005; Rousselle et al. 2008). Efficiency of the direct lighting calculation can be further improved by sampling from the *triple product* distribution of BSDF, illumination, and visibility; this issue was investigated by Ghosh and Heidrich (2006) and Clarberg and Akenine-Möller (2008b). Finally, Wang and Åkerlund (2009) have developed a technique that incorporates an approximation to the distribution of indirect illumination in the light sampling distribution used in these approaches.

The general idea of tracing light-carrying paths from the light sources was first investigated by Arvo (1986), who stored light in texture maps on surfaces and rendered caustics. Heckbert (1990b) built on this approach to develop a general ray-tracing-based global illumination algorithm, and Pattanaik and Mudur (1995) developed an early particle-tracing technique. Bidirectional path tracing was independently developed by Lafortune and Willems (1994) and Veach and Guibas (1994). Kollig and Keller (2000) developed the generalization of bidirectional path tracing to use quasi-random sample patterns.

The principles behind the virtual light source approach in the `IGIIntegrator` were first introduced by Keller's work on *instant radiosity* (1997). The instant global illumination algorithm was developed by Wald, Benthin, and collaborators (Wald et al. 2002, 2003; Benthin et al. 2003). The approach for compensating for bias from clamping the geometric term G that was implemented in Section 15.4.2 was developed by Kollig and Keller (2004). For scenes with complex lighting, many of the virtual lights may not make any contribution to the image being generated. To address this issue, Segovia and collaborators (2006, 2007) developed techniques based on bidirectional path tracing and Metropolis sampling to more efficiently generate virtual light sources. The virtual light source approach has recently been successfully used for real-time rendering of scenes with global illumination; see, for example, the papers by Dachsbacher and Stamminger (2006) and Nichols and Wyman (2009).

Building on the virtual point lights concept, Walter and collaborators (2005, 2006) developed *lightcuts*, which are based on creating thousands of virtual point lights and then building a hierarchy by progressively clustering nearby ones together. When a point is being shaded, traversal of the light hierarchy is performed by computing bounds on the error that would result from using clustered values to illuminate the point versus continuing down the hierarchy, leading to an approach with both guaranteed error bounds and good efficiency.

Ward and collaborators developed the irradiance caching algorithm, which is described in a series of papers (Ward, Rubinstein, and Clear 1988; Ward 1994b). One useful improvement to the basic form of the algorithm implemented here is an improved interpolation scheme described by Ward and Heckbert (1992); they estimated the change in irradiance at sample points as the point moves laterally over the surface and as the surface

IGIIntegrator 773

normal changes, based on the individual radiance samples used to compute the irradiance value. This technique results in a smoother interpolated irradiance function. Tabellion and Lamorlette (2004) described a number of additional improvements to irradiance caching that made it effective for rendering for movie productions; our implementation in this chapter generally follows their approach.

Křivánek and collaborators generalized irradiance caching to *radiance caching*, where a more complex directional distribution of incident radiance is stored, so that more accurate shading from glossy surfaces is possible (Křivánek et al. 2005a, 2005b). Gautron et al. (2007) showed how to effectively apply irradiance and radiance caching to rendering animations. Jarosz et al. (2008b) developed an improved gradient computation that better accounts for occlusion and the effect of participating media (i.e., the fact that radiance isn't constant along rays in scenes with volumetric scattering), and Herzog et al. (2009) showed how to use ray differentials for more accurate gradients. Gassenbauer et al. (2009) generalized the idea of caching radiance in the spatial domain to also cache values organized by direction, making the caching approach applicable to highly specular BSDFs as well as diffuse and glossy ones. See also the SIGGRAPH course notes on irradiance caching (Křivánek et al. 2007) and Křivánek and Gautron's book (2009) for a wealth of practical information about implementing these algorithms.

Approaches like Arvo's caustic rendering algorithm (Arvo 1986) formed the basis for an improved technique for caustics developed by Collins (1994). Jensen (1995, 1996a) developed the photon mapping algorithm, which introduced the key innovation of storing the light contributions in a general 3D data structure rather than in textures on surfaces. Important improvements to the photon mapping method are described in follow-up papers by Jensen and collaborators and Jensen's book on photon mapping (Jensen 1996b, 1997, 2001; Jensen and Christensen 1998). Density estimation techniques for global illumination were first introduced by Shirley, Walter, and collaborators (Shirley et al. 1995; Walter et al. 1997).

The question of how to find the most effective set of photons for photon mapping has seen significant amounts of research: light-driven particle-tracing algorithms don't work well for all scenes (consider, for example, a complex building model with lights in every room but where the camera sees only a single room). Early work in this area was done by Peter and Pietrek (1998) and Suykens and Willems (2000b), who developed techniques to store fewer photons in parts of the scene that are relatively unimportant. Fan et al. (2005) showed that the application of Veach's particle-tracing theory to photon mapping that was introduced in Section 15.6.1 demonstrates how photon paths starting from the camera can be generated. They were able to use this approach in conjunction with a Metropolis sampling algorithm to generate photon distributions that were more robust for complicated lighting configurations than those obtained from only following paths from light sources.

Given a set of photons, storing them efficiently is also important for a high-performance implementation. Jensen developed a memory-efficient representation for storing photons and for photon kd-tree nodes (Jensen 1996b), and Steinhurst et al. (2005) developed improved memory layout techniques to improve cache performance and reduce off-chip bandwidth when using photon mapping. Christensen and Batali (2004) developed techniques for storing illumination data for out-of-core use with complex scenes, where a

subset of the data is loaded from disk and cached in memory. Wald et al. (2004) demonstrated that a balanced kd-tree is not optimal for efficient photon lookups. They instead showed how to build kd-trees using a cost metric that estimates the traversal cost when searching for the k nearest neighbors in the tree, building on ideas for building efficient acceleration structures for ray tracing; they demonstrated that this approach led to significantly faster lookups.

Issues related to density estimation and how to best use available photons to compute reflected light have also received significant amounts of attention from researchers. Cammarano and Jensen (2002) showed how to extend photon mapping to support animated scenes. Hey and Purgathofer (2002) investigated density estimation techniques that accounted for the scene geometry where the lookup was being performed, and Schregle (2003) developed a method that chose the number of photons to use for density estimation based on a binary search that tried to find a number that made an optimal trade-off between noise and blurring (using too few and too many photons, respectively). Weber et al. (2004) applied bilateral filtering to photon map density estimates, and Havran et al. (2005a) developed techniques for storing the photon rays in space and interpolating them in this form, rather than storing their intersection points on surfaces. They showed that this approach addresses some of the reconstruction errors that occur when interpolating photons stored on surfaces.

Herzog and Seidel (2007) developed a density estimation kernel that accounts for variation in surface normal and occlusion effects; one source of error from conventional filtering approaches is that part of the area being filtered over may actually be occluded from the viewing perspective. Fabianowski and Dingliana (2009) developed a filtering method based on using ray differentials of the photons to adapt the kernel. Finally, rather than focusing on improved kernels, Spencer and Jones (2009b) had the insight that redistributing the photons so that no two of them are too close together makes it possible to use many fewer of them in estimates while still generating high-quality results.

One way to reduce the memory requirements for storing photons is to store few or none of them. To this end, Havran et al. (2005b) developed a final gathering algorithm based on storing final gather intersection points in a kd-tree in the scene and *then* shooting photons from the lights; when a photon intersects a surface, the nearby final gather intersection records are found and the photon's energy can be distributed to the origins of the corresponding final gather rays; the photon doesn't need to be stored at all. Herzog et al. (2007) described an approach based on storing all of the visible points as seen from the camera and *splatting* photon contributions to them, and Hachisuka et al. (2008b) developed the *progressive photon mapping* algorithm, where the points visible from the camera are stored and a series of iterations are executed where a fixed number of photons are traced and distributed to the points. After the points' reflected radiance values have been updated, the photons are discarded; as more iterations are performed, the values converge to the correct result.

Final gathering for finite-element radiosity algorithms was first described in Reichert's thesis (Reichert 1992). One important use of information available in the photon map is optimized final gathering techniques that importance-sample directions for the gathering step using the photons to give information about which directions are likely to

have large contributions (Jensen 1995; Hey and Purgathofer 2002). Christensen (1999) developed the important optimization for the final gathering step of precomputing outgoing radiance at a subset of the photons that was implemented here in Section 15.6.3. Arikan et al. (2005) described an efficient final gathering approach based on handling nearby and far-away components of the incident radiance field separately. More recently, Spencer and Jones (2009a) described how to build a hierarchical kd-tree of photons such that traversal could be stopped at higher levels of the tree and showed that using the footprints of final gather rays, computed using ray differentials, can lead to better results than the usual approach.

Veach and Guibas (1997) first applied the Metropolis sampling algorithm to solving the light transport equation. They demonstrated how this method could be applied to image synthesis and showed that the result was a light transport algorithm that was robust to traditionally difficult lighting configurations (e.g., light shining through a slightly ajar door). The approach used in the `MetropolisRenderer` to ensure that a given number of samples are proposed in each pixel is the same as the lens subpath mutation that they proposed. Kelemen et al. (2002) developed the formulation of Metropolis rendering used by the `MetropolisRenderer` in this chapter; their approach is more easily implemented than Veach and Guibas's. They also suggest a useful alternative approach for weighting the current and proposed samples based on multiple importance sampling. More recently, Fan et al. (2005) developed a method that let the user explicitly provide a number of important paths (for example, through a tricky geometric configuration) that could then be used in Metropolis sample generation. Cline et al. (2005) developed a light transport algorithm based on generating an independent chain of Metropolis samples in each pixel. Hoberock's Ph.D. dissertation discusses a number of alternatives for the scalar contribution function, including those that adapt the sampling density to pay more attention to particular modes of light transport and those that focus on reducing noise in the final image (Hoberock 2008).

Hair is particularly challenging to render; not only is it extremely geometrically complex, but multiple scattering among hair makes a significant contribution to its final appearance. Traditional light transport algorithms often have difficulty handling this case well. See the papers by Moon and Marschner (2006), Moon et al. (2008), and Zinke et al. (2008) for recent work in specialized rendering algorithms for hair.

EXERCISES

❷ 15.1 To further improve efficiency, Russian roulette can be applied to skip tracing many of the shadow rays that make a low contribution to the final image: to do so, tentatively compute the potential contribution of each shadow ray to the final overall radiance value before tracing the ray. If the contribution is below some threshold, apply Russian roulette to possibly skip tracing the ray. Recall that Russian roulette always increases variance; when evaluating the effectiveness of your implementation, you should consider its *efficiency*—how long it takes to render an image at a particular level of quality.

MetropolisRenderer 852

2 15.2 Read Veach's description of efficiency-optimized Russian roulette, which adaptively chooses a threshold for applying Russian roulette (Veach 1997; Section 10.4.1). Implement this algorithm in `pbrt` and evaluate its effectiveness in comparison to manually setting these thresholds.

3 15.3 Implement a technique for generating samples from the product of the light and BSDF distributions; see the papers by Burke et al. (2005), Ghosh et al. (2005), Cline et al. (2006), Clarberg et al. (2005), and Rousselle et al. (2008). Compare the effectiveness of the approach you implement to the direct lighting calculation currently implemented in `pbrt`. Investigate how scene complexity (and, thus, how expensive shadow rays are to trace) affects the Monte Carlo efficiency of the two techniques.

2 15.4 Clarberg and Akenine-Möller (2008b) described an algorithm that performs visibility caching, computing, and interpolating information about light source visibility at points in the scene. Implement this method and use it to improve the direct lighting calculation in `pbrt`. What sorts of scenes is it particularly effective for? Are there scenes for which it doesn't help?

2 15.5 Investigate algorithms for rendering scenes with large numbers of light sources: see, for example, the papers by Ward (1991a), Shirley, Wang, and Zimmerman (1996), Keller and Wald (2000), and Donikian et al. (2006) on this topic. Choose one of these approaches and implement it in `pbrt`. Run experiments with a number of scenes to evaluate the effectiveness of the approach that you implement.

3 15.6 Modify `pbrt` so that the user can flag certain objects in the scene as being important sources of indirect lighting and modify the `PathIntegrator` to sample points on those surfaces according to dA to generate some of the vertices in the paths it generates. Use multiple importance sampling to compute weights for the path samples, incorporating the probability that they would have been sampled both with BSDF sampling and with this area sampling. How much can this approach reduce variance and improve efficiency for scenes with substantial indirect lighting? How much can it hurt if the user flags surfaces that actually make little or no contribution, or if multiple importance sampling isn't used? Investigate generalizations of this approach that learn which objects are important sources of indirect lighting as rendering progresses so that the user doesn't need to supply this information ahead of time.

3 15.7 Light transport algorithms that trace paths from the lights like bidirectional path tracing and particle tracing implicitly assume that the BSDFs in the scene are symmetric—that $f(\mathrm{p}, \omega, \omega') = f(\mathrm{p}, \omega', \omega)$. However, both real-world BSDFs like the one describing specular transmission as well as synthetic BSDFs like the ones that are used when shading normals are present are actually not symmetric. Not accounting for this situation can lead to errors in the results computed by these light transport algorithms (Veach 1997; Chapter 5). Fix `pbrt` to properly handle nonsymmetric BSDFs. Demonstrate that these fixes eliminate errors that are present in the current implementation.

PathIntegrator 766

❸ 15.8 The implementation of the `IGIIntegrator` in Section 15.4 only traces paths starting from the light sources in order to place virtual point lights in the scene. This approach can be ineffective in scenes where most paths from the lights do not generate virtual lights that are visible from the points visible to the camera. Implement one of the improved approaches for placing virtual lights introduced by Segovia et al. (2006, 2007). Show the improvement in image quality that results for a scene where the `IGIIntegrator` doesn't do well.

❸ 15.9 Implement Walter et al.'s *lightcuts* algorithm in `pbrt` (Walter et al. 2005, 2006). How do the `BSDF` interfaces in `pbrt` need to be generalized to compute the error terms for light cuts? Do other core system interfaces need to be changed? Compare the efficiency of rendering images with global illumination using your implementation to some of the other integrators.

❷ 15.10 Read Ward and Heckbert's paper on irradiance gradients (Ward and Heckbert 1992) and implement this technique in the `IrradianceCacheIntegrator`. How much does it improve the quality of the results that it computes?

❷ 15.11 Improve the `IrradianceCacheIntegrator`'s computation of irradiance values by modifying it to use bidirectional path tracing instead of regular path tracing to compute the individual radiance values. Construct a scene where this approach is much better than path tracing. How much does it improve the results for scenes where the path-tracing approach works well already?

❸ 15.12 Modify `pbrt` to implement the irradiance filtering method described by Kontakten et al. (2004). Compare the results from this approach to the current `IrradianceCacheIntegrator` implementation.

❸ 15.13 Implement the radiance caching algorithm developed by Křivánek et al. (2005a, 2005b). Compare the results of rendering glossy indirect reflections with your implementation to the results from the current `IrradianceCacheIntegrator`: how much more accurate are the results compared with your implementation? Also compare your results to a reference solution computed with the `PathIntegrator` or `MetropolisRenderer`: how much more efficiently does your implementation produce these effects?

❶ 15.14 Experiment with the parameters to the `PhotonIntegrator` until you get a good feel for how they affect results in the final image. At a minimum, experiment with varying the search radius, the number of photons traced, and the number of photons used.

❷ 15.15 If photon mapping is being used only to render caustics in a scene, generation of the caustic map can be accelerated by flagging which objects in the scene potentially have specular components in their BSDFs and then building a directional table around each light source, recording which directions could potentially result in specular paths (Jensen 1996a). Photons can only be shot in these directions, saving the work of tracing photons in directions that are certain not to lead to caustics. Extend `pbrt` to support this technique. Note that different solutions will be needed for point light sources, directional light sources, and area

`IGIIntegrator` 773
`IrradianceCacheIntegrator` 786
`MetropolisRenderer` 852
`PathIntegrator` 766
`PhotonIntegrator` 802

light sources. How much does this approach end up improving performance for scenes where it's applicable?

❷ 15.16 Another approach to improving the efficiency of photon shooting is to start out by shooting photons from lights in all directions with equal probability, but then to dynamically update the probability of sampling directions based on which directions lead to light paths that have high throughput and which directions are less effective. Photons then must be reweighted based on the probability for shooting a photon in a particular direction. As long as there is always nonzero possibility of sending a photon in any direction, this approach doesn't introduce bias into the shooting algorithm. One advantage of this approach compared to the one in Exercise 15.15 is that the user of the renderer doesn't need to flag which objects may cast caustics ahead of time.

❸ 15.17 Section 15.6.1 introduces theory that leads to algorithms where photon paths can be generated starting from the camera as well as the lights; this approach was applied to photon mapping by Fan et al. (2005), who also used Metropolis sampling to select paths. Modify the `PhotonIntegrator` implementation to also generate paths from the camera, computing appropriate weights for them. Demonstrate the effectiveness of this approach for scenes where paths from the lights are not an efficient approach.

❸ 15.18 The photons around a point carry additional useful information about the illumination there that can be used for efficient rendering. See the ideas introduced by Keller and Wald (2000) and extend the photon mapping integrator to store direct lighting photons that also store which light source each direct lighting photon originated from. When computing direct illumination at a point, use the nearby direct lighting photons to choose light source sampling probabilities based on the estimated contribution of each of the light sources. Evaluate the effectiveness of this approach with a number of scenes.

❷ 15.19 Read the paper by Wald et al. (2004) on photon kd-tree construction and modify the implementation in `pbrt` to use their approach. How much difference in rendering time do you see when using their method with scenes that do photon mapping? (You may want to experiment with scenes with both smoothly varying indirect lighting as well as those with sharp caustics, which will exhibit relatively more variation in photon density.) Use a profiler to measure how much kd-tree construction time for the photon maps is affected by the new method as well as how the time spent performing photon lookups is affected.

❸ 15.20 Modify the `IrradianceCacheIntegrator` or `PhotonIntegrator` to use adaptive sampling for computing irradiance estimates or final gather contributions, respectively. Take some number of samples and use them to decide if the function being integrated seems to be relatively smooth or quickly changing. If it is smooth, stop sampling, but otherwise take a larger number of samples in an effort to compute a better estimate. Experiment with different criteria for deciding when to take more samples. How much of an efficiency improvement is your approach able to achieve compared to always taking a large number of

`IrradianceCacheIntegrator` 786
`PhotonIntegrator` 802

samples? Read Kirk and Arvo's paper that discusses how adaptive sampling can introduce bias (Kirk and Arvo 1991). In your implementation, is it worthwhile to use an unbiased adaptive sampling approach, or is the bias unobjectionable in practice?

● 15.21 Another approach for shooting photons that can *guarantee* constant weights is to use rejection sampling. Show how to use rejection sampling for shooting photons from lights and scattering photons from surfaces when the sampling distribution doesn't exactly match the function being sampled in a way that guarantees constant-weight photons. In what ways do the light source and BSDF sampling methods need to be modified for this approach?

● 15.22 Read Hachisuka et al.'s paper on progressive photon mapping (Hachisuka et al. 2008b) and implement their algorithm in `pbrt`. In order to maintain information about all of the points visible from the camera over the course of the entire rendering computation, you will likely need to implement their approach as a new `Renderer`. Compare the results with this algorithm to those generated by the regular `PhotonIntegrator` and the `MetropolisRenderer` for scenes with difficult lighting. Parallelize your implementation by breaking the various phases of computation into tasks. How well does performance scale with increasing numbers of processors?

● 15.23 The current implementation of the `MetropolisRenderer` always mutates all of the samples, up to the maximum ray depth that was specified. This approach is needlessly inefficient in that when the path length is shorter than the maximum depth (due to termination from Russian roulette or a ray leaving the scene), the extra samples are never used. However, generating samples lazily must be done carefully for correct results; for newly needed samples, it's necessary to replay the missed mutation history since the last large step. Read Kelemen et al.'s paper, which discusses this issue (Kelemen et al. 2002), and implement their lazy mutation strategy. As part of this work, make it possible for paths to have arbitrary lengths rather than a fixed maximum; doing so will require allowing both the `PathVertex` arrays as well as the `vectors` in the `MLTSample` to be able to grow dynamically.

● 15.24 By adding mutation strategies that don't necessarily modify all of the sample values X_i, it can be possible to reuse some or all of the paths generated by the previous samples in the `MetropolisRenderer`. For example, if only the `LightingSample` values are mutated in the `MLTSample`, then the camera and light paths can be reused and the shadow rays connecting their vertices don't need to be retraced. (Indeed, the `Spectrum` representing the entire indirect lighting contribution found from these connections can be reused.) In a similar manner, if only the light path's sample values are mutated, then not only does the camera path not need to be retraced, but the results from the direct lighting calculations at the camera path's vertices can be reused. As a third alternative, if the samples for the first few vertices of the camera or light path are left unchanged, then that portion of the path doesn't need to be retraced.

LightingSample 838
MetropolisRenderer 852
MLTSample 839
PathVertex 843

Modify the MetropolisRenderer to add one or more of the above sampling strategies and modify the implementation so that it reuses any partial results from the previous sample that remain valid when your new mutation is used. You may want to add both "small step" and "large step" variants of your new mutation. Compare the mean squared error of images rendered by your modified implementation to the MSE of images rendered by the original implementation, comparing to a reference image rendered with a large number of samples. For the same number of samples, your implementation should be faster but will likely have slightly higher error due to additional correlation between samples. Is the Monte Carlo efficiency of your modified version better than the original implementation?

● 15.25 In his Ph.D. dissertation, Hoberock proposes a number of alternative scalar contribution functions for Metropolis light transport, including ones that focus on particular types of light transport and ones that adapt the sample density during rendering in order to reduce perceptual error (Hoberock 2008). Read Chapter 6 of his thesis and implement either the multistage MLT or the noise-aware MLT algorithm that he describes.

● 15.26 A significant shortcoming of the bidirectional path-tracing algorithm implemented in the MetropolisRenderer is that it equally weights all of the various ways of generating a path of given length (recall the implementation in the fragment ⟨*Compute weight for bidirectional path,* pathWt⟩). Veach and Guibas demonstrated that applying multiple importance sampling to weighting bidirectional paths can lead to substantially less variance at relatively small incremental cost (Veach and Guibas 1995; Veach 1997). Modify the MetropolisRenderer to use MIS in its Lbidir() method and measure the improvement in image quality. How much does execution time increase from your change?

● 15.27 Another shortcoming of the bidirectional path-tracing algorithm implemented in the MetropolisRenderer is that it doesn't support paths that start at the light source and have zero camera path vertices. These types of paths are particularly important for rendering caustics. Consider, for example, caustics on a diffuse surface due to light passing through a glass sphere from a point light source: it's not possible to sample an outgoing direction from the diffuse surface that passes through the two delta-distribution BSDFs on the sphere and then intersects the point light—the only way to sample this type of path is to sample a path from the light and add contributions from points on the diffuse surface that are intersected by light paths and are visible to the camera.

Modify the MetropolisRenderer::Lbidir() method to support zero-vertex camera paths and modify the ImageFilm and Camera implementations to support adding contributions from arbitrary points in the scene to the image. (Contributions from points that aren't visible from the camera can just be discarded.) Don't forget to update the path weighting computation since your change will add one more way to generate a path of each length. You will also need to include the effect of scattering in the direction back toward the camera

Camera 302
ImageFilm 404
MetropolisRenderer 852
MetropolisRenderer::
Lbidir() 849

at the last vertex of the light path. One important detail is that with this change a single `MLTSample` may contribute to multiple pixels—one from the camera path and potentially one more for each vertex of the light path. Don't forget to add code to account for this in the `MLTTask::Run()` method.

Render images that show off the improvement from this change when rendering caustics. Does this approach help with efficiency for other indirect lighting cases?

MLTSample 839
MLTTask::Run() 857



CHAPTER SIXTEEN

★ 16 LIGHT TRANSPORT II: VOLUME RENDERING

Just as `SurfaceIntegrators` are the meeting point of scene geometry, materials, and lights, applying sophisticated algorithms to solve the light transport equation and determine the distribution of radiance in the scene, `VolumeIntegrators` are responsible for incorporating the effect of participating media (as described by `VolumeRegions`) into this process and determining how it affects the distribution of radiance. This chapter briefly introduces the equation of transfer, which describes how participating media change radiance along rays, and then describes the `VolumeIntegrator` interface as well as a few simple `VolumeIntegrator` implementations. Section 16.5 then describes the implementation of a surface integrator that accounts for the effect of *subsurface scattering*, where incident light travels some distance inside a surface before exiting. Although the approach is implemented as a `SurfaceIntegrator`, it is included in this chapter since its implementation is based on an approximate solution to light transport through participating media.

Many of the general approaches to light transport algorithms for surfaces described in the previous chapter can be extended to handle participating media as well. The "Further Reading" section and this chapter's exercises discuss these connections further.

16.1 THE EQUATION OF TRANSFER

The equation of transfer is the fundamental equation that governs the behavior of light in a medium that absorbs, emits, and scatters radiation (Chandrasekhar 1960). It accounts for all of the volume scattering processes described in Chapter 11—absorption, emission, and in- and out-scattering—to give an equation that describes the distribution of radiance in an environment. The light transport equation is in fact a special case of the equation of transfer, simplified by the lack of participating media and specialized for scattering from surfaces (Arvo 1993).

In its most basic form, the equation of transfer is an integro-differential equation that describes how the radiance along a beam changes at a point in space. It can be transformed into a pure integral equation that describes the effect of participating media from the infinite number of points along a line. It can be derived in a straightforward manner by subtracting the effects of the scattering processes that reduce energy along a beam (absorption and out-scattering) from the processes that increase energy along it (emission and in-scattering). Recall the source term from Section 11.1.4: it gives the change in radiance at a point p in a particular direction ω due to emission and in-scattered light from other points in the medium:

$$S(\mathrm{p}, \omega) = L_{\mathrm{ve}}(\mathrm{p}, \omega) + \sigma_{\mathrm{s}}(\mathrm{p}, \omega) \int_{\mathbb{S}^2} p(\mathrm{p}, -\omega' \to \omega) L_{\mathrm{i}}(\mathrm{p}, \omega') \, \mathrm{d}\omega'.$$

The source term accounts for all of the processes that add radiance to a ray.

The attenuation coefficient, $\sigma_{\mathrm{t}}(\mathrm{p}, \omega)$, accounts for all processes that reduce radiance at a point: absorption and out-scattering. The differential equation that describes its effect is

$$\mathrm{d}L_{\mathrm{o}}(\mathrm{p}, \omega) = -\sigma_{\mathrm{t}}(\mathrm{p}, \omega) L_{\mathrm{i}}(\mathrm{p}, -\omega) \mathrm{d}t.$$

The overall differential change in radiance at a point p' along a ray is found by adding these two effects together to get the integro-differential form of the equation of transfer:[1]

$$\frac{\partial}{\partial t} L_{\mathrm{o}}(\mathrm{p}, \omega) = -\sigma_{\mathrm{t}}(\mathrm{p}, \omega) L_{\mathrm{i}}(\mathrm{p}, -\omega) + S(\mathrm{p}, \omega).$$

With suitable boundary conditions, this equation can be transformed to a purely integral equation. For example, if we assume that there are no surfaces in the scene so that the rays are never blocked and have an infinite length, the integral equation of transfer is

$$L_{\mathrm{i}}(\mathrm{p}, \omega) = \int_0^\infty T_r(\mathrm{p}' \to \mathrm{p}) S(\mathrm{p}', -\omega) \, \mathrm{d}t,$$

where $\mathrm{p}' = \mathrm{p} + t\omega$ (Figure 16.1). The meaning of this equation is reasonably intuitive: it just says that the radiance arriving at a point from a given direction is contributed to by the added radiance along all points along the ray from the point. The amount of added radiance at each point along the ray that reaches the ray's origin is reduced by the total beam transmittance from the ray's origin to the point.

More generally, if there are reflecting and/or emitting surfaces in the scene, rays don't necessarily have infinite length and the surface that a ray hits affects its radiance, adding outgoing radiance from the surface at the point and preventing radiance from points along the ray beyond the intersection point from contributing to radiance at the ray's origin. If a ray (p, ω) intersects a surface at some point p_0 a distance t along the ray, then the integral equation of transfer is

$$L_{\mathrm{i}}(\mathrm{p}, \omega) = T_r(\mathrm{p}_0 \to \mathrm{p}) L_{\mathrm{o}}(\mathrm{p}_0, -\omega) + \int_0^t T_r(\mathrm{p}' \to \mathrm{p}) S(\mathrm{p}', -\omega) \mathrm{d}t', \qquad [16.1]$$

1 It is an integro-differential equation due to the integral over the sphere in the source term.

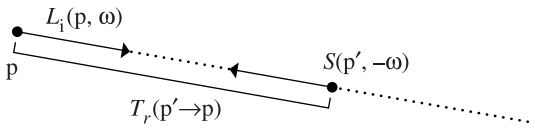


Figure 16.1: The equation of transfer gives the incident radiance at point $L_i(p, \omega)$ accounting for the effect of participating media. At each point along the ray, the source term $S(p', \omega)$ gives the differential radiance added at the point due to scattering and emission. This radiance is then attenuated by the beam transmittance $T_r(p' \to p)$ from the point p' to the ray's origin.

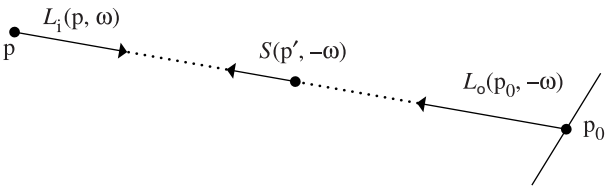


Figure 16.2: For a finite ray that intersects a surface, the incident radiance, $L_i(p, \omega)$, is equal to the outgoing radiance from the surface, $L_o(p_0, -\omega)$, times the beam transmittance to the surface plus the added radiance from all points along the ray from p to p_0.

where $p_0 = p + t\omega$ is the point on the surface and $p' = p + t'\omega$ are points along the ray (Figure 16.2).

This equation describes the two effects that contribute to radiance along the ray. First, reflected radiance back along the ray from the surface is given by the L_o term, which gives the emitted and reflected radiance from the surface. This radiance may be attenuated by the participating media; the beam transmittance from the ray origin to the point p_0 accounts for this. The second term accounts for the added radiance along the ray due to volume scattering and emission, but only up to the point where the ray intersects the surface; points beyond that one don't affect the radiance along the ray.

In the interest of brevity, we will refrain from further generalization of the equation of transfer here. However, just as the light transport equation could be written in a more general form as a sum over paths of various numbers of vertices and just as an importance function could be introduced to turn it into the measurement equation, the equation of transfer can be generalized in a similar manner. Likewise, we will only present a few simple `VolumeIntegrators` in most of this chapter, although the general types of algorithms used for the surface integrators in the previous chapter, such as path tracing, bidirectional path tracing, photon mapping, and so on, can be applied to volume integration as well.

VolumeIntegrator 876

16.2 VOLUME INTEGRATOR INTERFACE

The VolumeIntegrator interface inherits from Integrator, picking up the Preprocess(), and RequestSamples() methods declared there. These two methods are used by volume integrators in the same way as by surface integrators.

VolumeIntegrator adds a Li() method that is similar to the surface integrator version in that it returns the radiance along the given ray; volume integrators should assume that if the given ray does intersect a surface in the scene, then its Ray::maxt value will have been set to be at the intersection point. As such, the volume integrator should only compute the effect of volume scattering from the parametric range [mint, maxt] along the ray. The VolumeIntegrator's version of Li() doesn't include the Intersection parameter that SurfaceIntegrator has, and it adds an output Spectrum parameter to return the transmittance from the start of the ray to the end of the ray. This returned transmittance value is used to compute the attenuation of outgoing radiance from the intersected surface. Like SurfaceIntegrator::Li(), the Sample pointer may be NULL, in which case the implementation should use the provided RNG for any random sampling it performs.

⟨*VolumeIntegrator Interface*⟩ ≡
```
virtual Spectrum Li(const Scene *scene, const Renderer *renderer,
    const RayDifferential &ray, const Sample *sample, RNG &rng,
    Spectrum *transmittance, MemoryArena &arena) const = 0;
```

The VolumeIntegrator interface adds an additional method that implementations must provide, Transmittance(), which is responsible for computing the beam transmittance along the given ray from Ray::mint to Ray::maxt. This method is used to compute attenuation along shadow rays traced for direct lighting, for example.

⟨*VolumeIntegrator Interface*⟩ +≡
```
virtual Spectrum Transmittance(const Scene *scene,
    const Renderer *renderer, const RayDifferential &ray,
    const Sample *sample, RNG &rng, MemoryArena &arena) const = 0;
```

With this background, the SamplerRenderer::Li() method can be fully understood. It is a direct implementation of Equation (16.1). The surface integrator computes outgoing radiance L_o at the ray's intersection point, ignoring attenuation back to the ray origin. The volume integrator's Li() method gives the radiance along the ray due to participating media and also returns the beam transmittance T_r to the point on the surface. The sum of $L_o T_r$ and the additional radiance from participating media gives the total radiance arriving at the ray origin.

Integrator 740
Intersection 186
MemoryArena 1015
Ray::maxt 67
Ray::mint 67
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
SamplerRenderer::Li() 34
Scene 22
Spectrum 263
SurfaceIntegrator 740
SurfaceIntegrator::Li() 741
VolumeIntegrator 876

16.3 EMISSION-ONLY INTEGRATOR

The simplest possible volume integrator (other than one that ignored participating media completely) simplifies the source term by ignoring in-scattering and only accounting for emission and attenuation. Because in-scattering is ignored, the integral over the sphere in the source term at each point along the ray disappears, and the resulting sim-

plified equation of transfer is

$$L_i(p, \omega) = T_r(p_0 \to p) L_o(p_0, -\omega) + \int_0^t T_r(p' \to p) L_{ve}(p', -\omega)\, dt. \quad [16.2]$$

The `EmissionIntegrator`, which is defined in the files `integrators/emission.h` and `integrators/emission.cpp`, uses Monte Carlo integration to solve this equation. Figure 16.3(a) shows the ecosystem scene without any participating media; Figure 16.3(b), with fog rendered with the `EmissionIntegrator`.

⟨*EmissionIntegrator Declarations*⟩ ≡
```
class EmissionIntegrator : public VolumeIntegrator {
public:
    ⟨EmissionIntegrator Public Methods 877⟩
private:
    ⟨EmissionIntegrator Private Data 877⟩
};
```

The `EmissionIntegrator`'s `Transmittance()` and `Li()` methods both have to evaluate one-dimensional integrals along points t' along a ray. Rather than using a fixed number of samples for each estimate, the implementation here bases the number of samples on the distance the ray travels in the volume—the longer the distance, the more samples are taken. This approach is worthwhile for naturally intuitive reasons—the longer the ray's extent in the medium, the more accuracy is desirable and the more samples are likely to be needed to capture greater variation in optical properties along the ray. The number of samples taken is determined indirectly by a user-supplied parameter giving a step size between samples. The ray is divided into segments of the given length, and a single sample is taken in each one.

⟨*EmissionIntegrator Public Methods*⟩ ≡ 877
```
EmissionIntegrator(float ss) { stepSize = ss; }
```

⟨*EmissionIntegrator Private Data*⟩ ≡ 877
```
float stepSize;
```

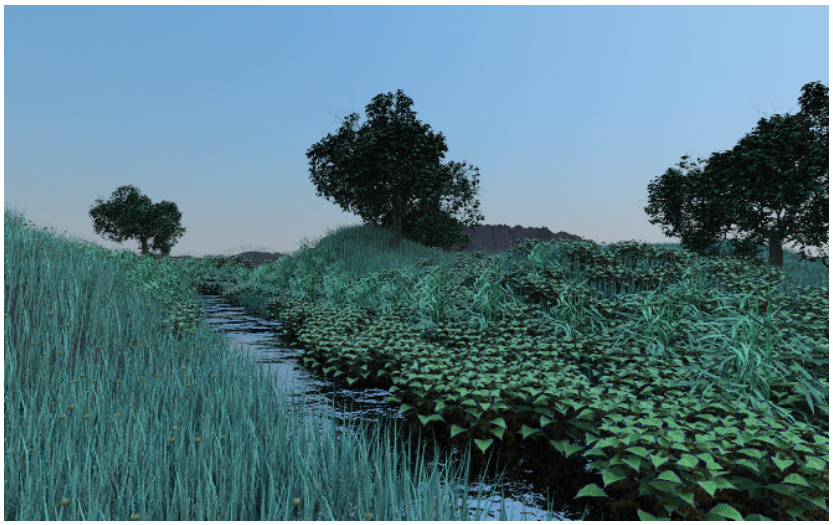
The `Transmittance()` and `Li()` methods each only need a single 1D sample value to evaluate their respective integrals:

⟨*EmissionIntegrator Method Definitions*⟩ ≡
```
void EmissionIntegrator::RequestSamples(Sampler *sampler, Sample *sample,
                                        const Scene *scene) {
    tauSampleOffset = sample->Add1D(1);
    scatterSampleOffset = sample->Add1D(1);
}
```

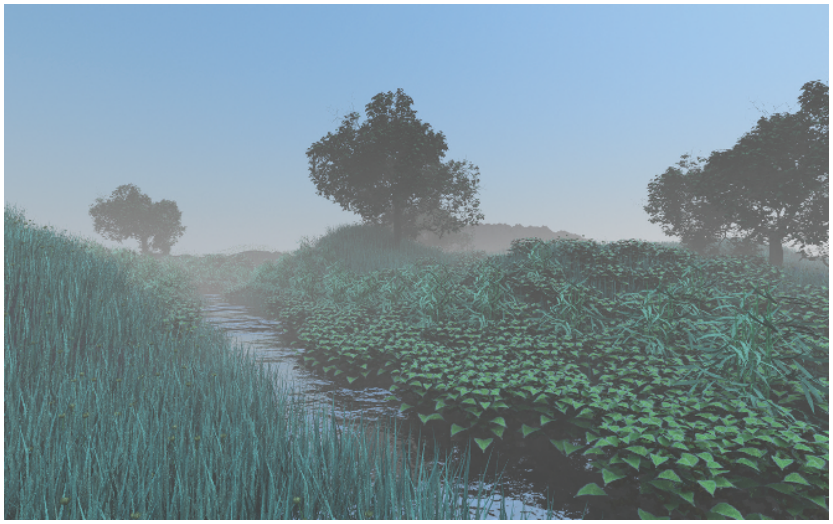
⟨*EmissionIntegrator Private Data*⟩ +≡ 877
```
int tauSampleOffset, scatterSampleOffset;
```

The `Transmittance()` method is reasonably straightforward. The `VolumeRegion`'s `tau()` method takes care of computing the optical thickness τ from the ray's starting point to its ending point. The only work for the integrator here is to choose a step size (in case `tau()` does Monte Carlo integration, as the implementation in Section 14.7 does), pass a

EmissionIntegrator 877
EmissionIntegrator::scatterSampleOffset 877
EmissionIntegrator::stepSize 877
EmissionIntegrator::tauSampleOffset 877
Sample 343
Sample::Add1D() 344
Sampler 340
Scene 22
VolumeIntegrator 876
VolumeRegion 587



(a)



(b)

Figure 16.3: Ecosystem scene rendered (a) without any participating media and (b) with `ExponentialDensity`-based fog rendered with the `EmissionIntegrator`. Participating media makes the image look substantially more realistic by reducing the contrast of faraway objects.

single sample value to that method, and return $e^{-\tau}$. If the `tau()` method can compute τ analytically, it will ignore these additional values.

`EmissionIntegrator` 877

`ExponentialDensity` 595

This method takes advantage of the fact that the `Sample` value is only non-`NULL` for camera rays to increase the step size for shadow and indirect rays, thus reducing computa-

tional demands (and accuracy). The reduction in accuracy for those rays generally isn't noticeable.

⟨*EmissionIntegrator Method Definitions*⟩ +≡

```
Spectrum EmissionIntegrator::Transmittance(const Scene *scene,
        const Renderer *renderer, const RayDifferential &ray,
        const Sample *sample, RNG &rng, MemoryArena &arena) const {
    if (!scene->volumeRegion) return Spectrum(1.f);
    float step, offset;
    if (sample) {
        step = stepSize;
        offset = sample->oneD[tauSampleOffset][0];
    }
    else {
        step = 4.f * stepSize;
        offset = rng.RandomFloat();
    }
    Spectrum tau = scene->volumeRegion->tau(ray, step, offset);
    return Exp(-tau);
}
```

The `Li()` method is responsible for evaluating the second term of the sum in Equation (16.2). If the ray enters the volume at $t = t_0$ along the ray, then no attenuation or emission happens from the start of the ray up to t_0, and the `Li()` method can instead consider the integral from t_0 to t_1, where t_1 is the minimum of the parametric offset where the ray exits the volume and the offset where it intersects a surface (Figure 16.4). An estimate of the value of this integral,

$$\int_{t_0}^{t_1} T_r(\mathrm{p}' \to \mathrm{p}) L_{\mathrm{ve}}(\mathrm{p}', -\omega)\, \mathrm{d}t',$$

EmissionIntegrator 877
EmissionIntegrator::stepSize 877
EmissionIntegrator::tauSampleOffset 877
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
RNG::RandomFloat() 1003
Sample 343
Sample::oneD 344
Scene 22
Scene::volumeRegion 23
Spectrum 263
Spectrum::Exp() 265
VolumeRegion::tau() 588

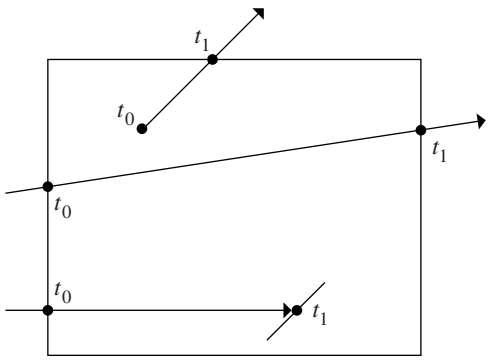


Figure 16.4: The starting point of the ray marching process, t_0, is the maximum of the ray's origin or the point where the ray enters the participating medium. Similarly, the end, t_1, is the minimum of the point where the ray exits the medium and the point where it intersects a surface, if any.

can be found by uniformly selecting random points p_i along the ray between t_0 and t_1 and evaluating the estimator

$$\frac{1}{N}\sum_i \frac{T_r(\mathrm{p}_i \to \mathrm{p})L_{\mathrm{ve}}(\mathrm{p}_i, -\omega)}{p(\mathrm{p}_i)} = \frac{t_1 - t_0}{N}\sum_i T_r(\mathrm{p}_i \to \mathrm{p})L_{\mathrm{ve}}(\mathrm{p}_i, -\omega)$$

since for uniformly sampled points $p(\mathrm{p}_i) = 1/(t_1 - t_0)$. The L_{ve} term in the estimator can be evaluated directly with the corresponding `volumeRegion` method, and the optical thickness τ to evaluate T_r can be evaluated either directly (for a homogeneous or exponential atmosphere), or via Monte Carlo integration as described in Section 14.7.

In order to do this computation, the implementation of the `Li()` method thus starts out by finding the t range for the integral and initializing `t0` and `t1` accordingly:

⟨*EmissionIntegrator Method Definitions*⟩ +≡
```
Spectrum EmissionIntegrator::Li(const Scene *scene,
        const Renderer *renderer, const RayDifferential &ray,
        const Sample *sample, RNG &rng, Spectrum *T,
        MemoryArena &arena) const {
    VolumeRegion *vr = scene->volumeRegion;
    float t0, t1;
    if (!vr || !vr->IntersectP(ray, &t0, &t1)) {
        *T = Spectrum(1.f);
        return 0.f;
    }
    ⟨Do emission-only volume integration in vr 880⟩
}
```

Two additional techniques are used in the implementation here. First, just as the `VolumeRegion::tau()` method in Section 14.7 used a uniform step size between sample points, this implementation also steps uniformly for similar reasons. Second, the beam transmittance values T_r can be evaluated efficiently if the points p_i are sorted from the one closest to the ray origin p to the one farthest away. Then, the multiplicative property of T_r can be used to incrementally compute T_r from its value from the previous point:

$$T_r(\mathrm{p}_i \to \mathrm{p}) = T_r(\mathrm{p}_i \to \mathrm{p}_{i-1})T_r(\mathrm{p}_{i-1} \to \mathrm{p}).$$

Because $T_r(\mathrm{p}_i \to \mathrm{p}_{i-1})$ covers a shorter distance than $T_r(\mathrm{p}_i \to \mathrm{p})$, fewer samples can be used to estimate its value if it is evaluated with Monte Carlo. Both of these ideas are illustrated in Figure 16.5.

⟨*Do emission-only volume integration in* vr⟩ ≡ 880
```
Spectrum Lv(0.);
⟨Prepare for volume integration stepping 881⟩
for (int i = 0; i < nSamples; ++i, t0 += step) {
    ⟨Advance to sample at t0 and update T 881⟩
    ⟨Compute emission-only source term at p 882⟩
}
*T = Tr;
return Lv * step;
```

EmissionIntegrator 877
MemoryArena 1015
RayDifferential 69
Renderer 24
RNG 1003
Sample 343
Scene 22
Scene::volumeRegion 23
Spectrum 263
VolumeRegion 587
VolumeRegion::IntersectP() 588
VolumeRegion::tau() 588

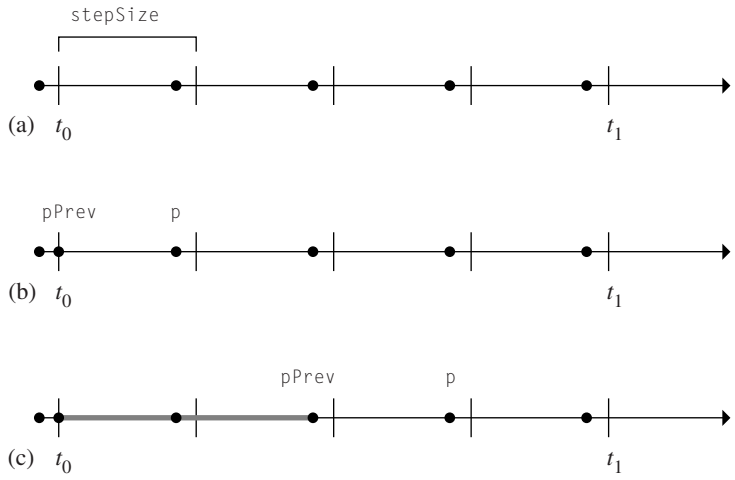


Figure 16.5: Volume Integration Ray Sampling. (a) The ray's extent from t_0 to t_1 is subdivided into a number of segments based on the user-supplied `stepSize` parameter. A single sample is taken in each segment, where the first sample is placed randomly in the first segment and all additional samples are offset by equal-sized steps. (b) Ray marching tracks the previous point to which transmittance was computed, `pPrev`, as well as the current point, `p`. Initially, `pPrev` is the point where the ray enters the volume. (c) At each subsequent step, beam transmittance is computed as the product of transmittance to `pPrev` and the additional transmittance from `pPrev` to `p`.

⟨*Prepare for volume integration stepping*⟩ ≡ 880
```
int nSamples = Ceil2Int((t1-t0) / stepSize);
float step = (t1 - t0) / nSamples;
Spectrum Tr(1.f);
Point p = ray(t0), pPrev;
Vector w = -ray.d;
t0 += sample->oneD[scatterSampleOffset][0] * step;
```

Ceil2Int() 1002
EmissionIntegrator::
scatterSampleOffset 877
EmissionIntegrator::
stepSize 877
Point 63
Ray 66
Ray::depth 67
Ray::time 67
RNG::RandomFloat() 1003
Sample::oneD 344
Spectrum 263
Spectrum::Exp() 265
Vector 57
VolumeRegion::tau() 588

To find the overall transmittance at the current point, it's only necessary to find the transmittance from the previous point to the current point and multiply it by the transmittance from the ray origin to the previous point, as described earlier.

⟨*Advance to sample at* `t0` *and update* `T`⟩ ≡ 880
```
pPrev = p;
p = ray(t0);
Ray tauRay(pPrev, p - pPrev, 0.f, 1.f, ray.time, ray.depth);
Spectrum stepTau = vr->tau(tauRay,
                           .5f * stepSize, rng.RandomFloat());
Tr *= Exp(-stepTau);
⟨Possibly terminate ray marching if transmittance is small 882⟩
```

In a thick medium, the transmittance may become very low after the ray has passed a sufficient distance through it. To reduce the time spent computing source term values

that are likely to have very little contribution to the radiance at the ray's origin, ray stepping is randomly terminated with Russian roulette when transmittance is sufficiently small.

⟨*Possibly terminate ray marching if transmittance is small*⟩ ≡ 881
```
if (Tr.y() < 1e-3) {
    const float continueProb = .5f;
    if (rng.RandomFloat() > continueProb) break;
    Tr /= continueProb;
}
```

Having done all this work, computing the source term at the point is easy:

⟨*Compute emission-only source term at* p⟩ ≡ 880
```
Lv += Tr * vr->Lve(p, w, ray.time);
```

16.4 SINGLE SCATTERING INTEGRATOR

In addition to accounting for the emission at each point along the ray, the `SingleScatteringIntegrator` also considers the incident radiance due to direct illumination but ignores incident radiance due to multiple scattering. Thus, its `Li()` method estimates the integral

$$\int_0^t T_r(\mathrm{p}' \to \mathrm{p}) \left(L_{\mathrm{ve}}(\mathrm{p}', -\omega) + \sigma_{\mathrm{s}}(\mathrm{p}', \omega) \int_{\mathbb{S}^2} p(\mathrm{p}', -\omega' \to -\omega) L_{\mathrm{d}}(\mathrm{p}', \omega') \mathrm{d}\omega' \right) \mathrm{d}t',$$

where L_{d} only includes radiance from direct lighting. This radiance may be blocked by geometry in the scene and may itself be attenuated by participating media between the light and the point p' along the ray (Figure 16.6). Although accounting for this effect can be much more computationally expensive than just ignoring it as the `EmissionIntegrator` does, it can give striking "beams of light" effects, as seen in Figures 11.1 and 16.7.

Almost all of the implementation of the `SingleScatteringIntegrator` parallels the `EmissionIntegrator`, so only the fragments in the parts that differ are included here. This integrator is defined in the files `integrators/single.h` and `integrators/single.cpp`.

⟨*SingleScatteringIntegrator Declarations*⟩ ≡
```
class SingleScatteringIntegrator : public VolumeIntegrator {
public:
    ⟨SingleScatteringIntegrator Public Methods⟩
private:
    ⟨SingleScatteringIntegrator Private Data⟩
};
```

This integrator's `Li()` method uses the same general ray-marching approach to evaluate the equation of transfer as the `EmissionIntegrator`. One difference is that this one computes sample values for light source sampling before it enters the `for` loop over sample positions. Because it isn't known how many samples will be necessary until `Li()` is called

EmissionIntegrator 877
RNG::RandomFloat() 1003
SingleScatteringIntegrator 882
Spectrum::y() 273
VolumeIntegrator 876
VolumeRegion::Lve() 588

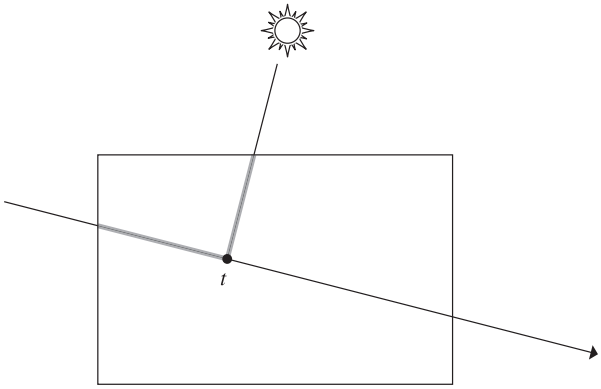


Figure 16.6: When the direct lighting contribution is evaluated at some point t along a ray passing through participating media, it's necessary to compute the attenuation of the radiance from the light passing through the volume to the scattering point as well as the attenuation from that point back to the ray origin.

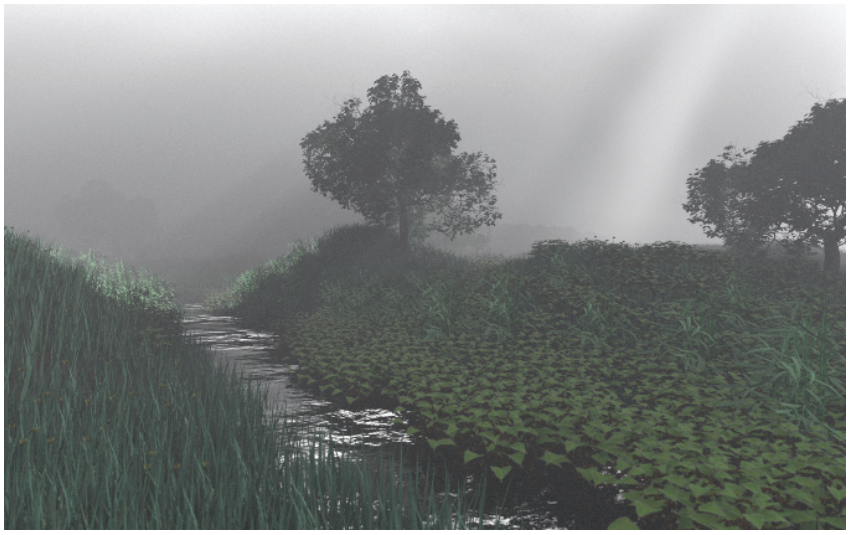


Figure 16.7: When the single scattering volume integrator is used with the ecosystem scene, beams of light are visible, giving a striking visual effect.

(since this number depends on the length of the ray segment over which integration is being done), it's not possible to have the `Sampler` generate samples and pass them into `Li()` via the `Sample`. Therefore, the samples are generated here. Two one-dimensional patterns are used for selecting which light to sample and which light component to sample, and a two-dimensional pattern is used for selecting points on area light sources.

⟨*Compute sample patterns for single scattering samples*⟩ ≡
```
float *lightNum = arena.Alloc<float>(nSamples);
LDShuffleScrambled1D(1, nSamples, lightNum, rng);
float *lightComp = arena.Alloc<float>(nSamples);
LDShuffleScrambled1D(1, nSamples, lightComp, rng);
float *lightPos = arena.Alloc<float>(2*nSamples);
LDShuffleScrambled2D(1, nSamples, lightPos, rng);
uint32_t sampOffset = 0;
```

The other difference from the `EmissionIntegrator` is how the source term is evaluated. At each sample point along the ray, the following fragment computes the single scattering approximation of the source term at the point `p`. It serves the same role as the earlier ⟨*Compute emission-only source term at* `p`⟩ fragment. After including volume emission in the same way that the `EmissionIntegrator` does, it finds the value of σ_s at the point, selects a light to sample, and computes its contribution to scattering at the point. Because the source term is generally evaluated at many points along the ray, only a single light is sampled at each one, and its contribution is scaled by the number of lights, similar to the direct lighting integrator's "sample one light" strategy.

⟨*Compute single-scattering source term at* `p`⟩ ≡
```
Lv += Tr * vr->Lve(p, w, ray.time);
Spectrum ss = vr->sigma_s(p, w, ray.time);
if (!ss.IsBlack() && scene->lights.size() > 0) {
    int nLights = scene->lights.size();
    int ln = min(Floor2Int(lightNum[sampOffset] * nLights),
                 nLights-1);
    Light *light = scene->lights[ln];
    ⟨Add contribution of light due to scattering at p 884⟩
}
++sampOffset;
```

Computing the estimate of the direct lighting contribution at a point p involves estimating the integral

$$\sigma_s(\mathrm{p}, \omega) \int_{S^2} p(\mathrm{p}, -\omega' \rightarrow -\omega) L_d(\mathrm{p}, \omega')\, d\omega'.$$

Rather than sampling both the phase function and the light source and applying multiple importance sampling, the implementation here always lets the light choose a sample position on the light source and then computes the estimator directly. For media that aren't extremely anisotropic, this approach works well.

⟨*Add contribution of* `light` *due to scattering at* `p`⟩ ≡ 884
```
float pdf;
VisibilityTester vis;
Vector wo;
LightSample ls(lightComp[sampOffset], lightPos[2*sampOffset],
               lightPos[2*sampOffset+1]);
Spectrum L = light->Sample_L(p, 0.f, ls, ray.time, &wo, &pdf, &vis);
```

Floor2Int() 1002
LDShuffleScrambled1D() 377
LDShuffleScrambled2D() 377
Light 606
Light::Sample_L() 608
LightSample 710
MemoryArena::Alloc() 1016
Scene::lights 23
Spectrum 263
Spectrum::IsBlack() 265
Vector 57
VisibilityTester 608
VisibilityTester::
Transmittance() 609
VisibilityTester::
Unoccluded() 609
VolumeRegion::Lve() 588
VolumeRegion::p() 588
VolumeRegion::sigma_s() 588

```
if (!L.IsBlack() && pdf > 0.f && vis.Unoccluded(scene)) {
    Spectrum Ld = L * vis.Transmittance(scene, renderer, NULL, rng, arena);
    Lv += Tr * ss * vr->p(p, w, -wo, ray.time) * Ld * float(nLights) /
            pdf;
}
```

16.5 SUBSURFACE SCATTERING

The bidirectional scattering-surface reflectance distribution function (BSSRDF) was introduced in Section 5.6.2; it gives exitant differential radiance at a point on a surface $\mathrm{p_o}$ given incident differential irradiance at another point $\mathrm{p_i}$: $S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)$. Accurately rendering translucent surfaces with subsurface scattering requires integrating over both area—points on the surface of the object being rendered—and incident direction, evaluating the BSSRDF and computing reflection with the subsurface scattering equation

$$L_o(\mathrm{p_o}, \omega_o) = \int_A \int_{\mathcal{H}^2(\mathbf{n})} S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)\, L_i(\mathrm{p_i}, \omega_i) |\cos \theta_i|\, \mathrm{d}\omega_i\, \mathrm{d}A.$$

Figure 16.8 suggests the complexity of evaluating the integral. To compute the standard Monte Carlo estimate of this equation given a point at which to compute outgoing radiance, we need a technique to sample points $\mathrm{p_i}$ on the surface and to compute the incident radiance at these points, as well as an efficient way to compute the specific value of the BSSRDF $S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)$ for each sampled point $\mathrm{p_i}$ and incident direction.

For translucent materials, the scattering properties inside the material are described by the volume scattering mechanisms introduced in Section 11.1. The rendering problem is thus to take an irregularly shaped object where the boundary defines the extent of a

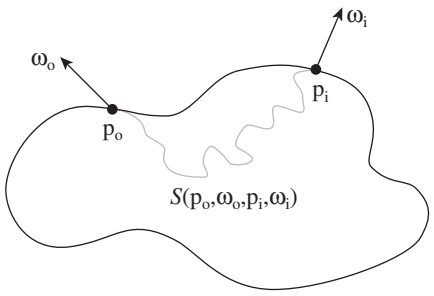


Figure 16.8: Computing Subsurface Reflection. When a surface is translucent, in order to compute outgoing radiance from a point $\mathrm{p_o}$ in direction ω_o, it's necessary to sample a number of points $\mathrm{p_i}$ and compute incident radiance at them. Then, the BSSRDF, $S(\mathrm{p_o}, \omega_o, \mathrm{p_i}, \omega_i)$, must be evaluated in order to determine the effect of incident radiance at one point on outgoing radiance at the point of interest. The BSSRDF can be difficult to evaluate efficiently, since it represents all scattering within the volume for light that enters at one point and exits at the other.

medium with these scattering properties, and to render images of the object accounting for the light transport in the medium. Thus, the BSSRDF S notationally represents the end effect of these scattering processes between a given pair of points and directions on the boundary.

A multiple scattering volume integrator can be used to evaluate the BSSRDF: given a pair of points on the surface and a pair of directions, the integrator can be used to compute the fraction of incident light from direction ω_i at the point p_i that exits the object at the point p_o in direction ω_o by following light-carrying paths through the multiple scattering events in the medium. Beyond standard path-tracing or bidirectional path-tracing techniques, many existing light transport algorithms are applicable to this task; for example, Jensen and Christensen (1998) applied volume photon maps.

However, many translucent objects are characterized by having very high albedos, which are not efficiently handled by classic approaches. For example, Jensen et al. (2001b) measured the scattering properties of skim milk and found an albedo of 0.9987. When essentially all of the light is scattered at each interaction in the medium and almost none of it is absorbed, light easily travels far from where it first enters the medium. Hundreds or even thousands of scattering events must be considered to compute an accurate result; given the high albedo of milk, after 100 scattering events, 87.5% of the incident light is still carried by a path, 51% after 500 scattering events, and still 26% after 1000.

Efficiently rendering these kinds of interesting translucent scattering media therefore requires a different approach to light transport than we've used so far. In this section, we'll present the implementation of an efficient method for rendering the effect of subsurface scattering in high-albedo materials. Figure 16.9 shows an example of the dragon model rendered with this integrator. Note that although light is arriving from above and slightly behind it, the effect of light traveling inside the medium and exiting far from where it entered is well modeled.

The method we'll implement in this section is based on two main ideas. First, the distribution of light in the translucent medium is modeled with the *diffusion approximation*, which describes the equlibrium distribution of illumination in highly scattering optically thick participating media. Second, a solution to the diffusion equation is found by composing closed-form solutions of it for point light sources; a *dipole* of two light sources is used to approximate the overall scattering.

This approach makes a number of simplifications—it assumes heterogeneous scattering properties throughout the medium, and it implicitly assumes that the medium is *semi-infinite* (it extends infinitely beneath a planar surface). In practice, however, it can still accurately render many interesting surfaces. The "Further Reading" section and exercises at the end of this chapter have references to improvements to this approach that generalize it to handle more general settings more accurately.

Our implementation of subsurface scattering has three main components, each of which will be described in its own section to follow:

1. A large number of random points on the surfaces of the translucent objects in the scene are sampled and incident irradiance is computed at these points.
2. A hierarchical representation of these points is created, progressively clustering nearby groups of them summing their irradiance values.

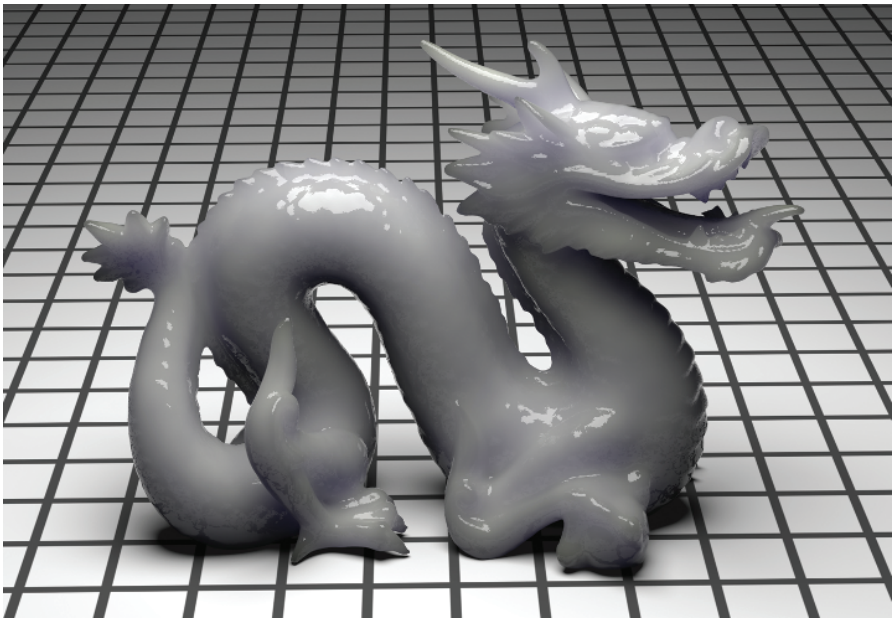


Figure 16.9: Subsurface Scattering from the Dragon Model. Although incident illumination is arriving from behind the model, the front of the model has light exiting from it, due to subsurface light transport.

3. At rendering time, these points are used in a hierarchical integration algorithm that evaluates the subsurface scattering equation at the point being shaded. To evaluate the integrand, it uses an approximation based on the dipole diffusion approximation that efficiently evaluates the BSSRDF for a given pair of points on the surface.

The `DipoleSubsurfaceIntegrator` class implements this approach. It is implemented in the files `integrators/dipolesubsurface.h` and `integrators/dipolesubsurface.cpp`. Its constructor takes parameters that set the maximum depth for specular reflections and refractions, a maximum error control (its use will be explained later in this section), a minimum distance that controls the density of points on the surfaces of objects for the first step above, and an optional filename. If the filename isn't an empty string, then a collection of precomputed points on (presumably translucent) surfaces is read from the given file. Otherwise, a set of points is found during a preprocessing step.

⟨*DipoleSubsurfaceIntegrator Public Methods*⟩ ≡

```
DipoleSubsurfaceIntegrator(int mdepth, float merror, float mindist,
                           const string &fn) {
    maxSpecularDepth = mdepth;
    maxError = merror;
    minSampleDist = mindist;
    filename = fn;
}
```

⟨*DipoleSubsurfaceIntegrator Private Data*⟩ ≡
```
int maxSpecularDepth;
float maxError, minSampleDist;
string filename;
```

16.5.1 POISSON DISTRIBUTION OF SAMPLE POINTS

For the hierarchical integration algorithm, we need to generate a collection of points on the surfaces of the translucent objects in the scene at which we'll compute incident irradiance. These points should be reasonably uniformly distributed so that they sample the irradiance at a regular rate; if there were clumps of multiple points close together, the hierarchical integration algorithm would compute poor results.

A Poisson disk distribution of points on the surface would be a good distribution for integration. (Recall the 2D Poisson disk sampling pattern for images in Section 7.5, though there the points weren't on object surfaces.) A number of approaches to directly compute such a distribution are available, though all require a more complex set of `Shape` interfaces than `pbrt` provides. (For example, if a parameterization of the surface is available or if the geometry is constrained to be a triangle mesh, *point repulsion* algorithms can be used, where a number of points are placed randomly on the surface and then their positions are iteratively adjusted until none of them are too close together.) Rather than requiring `Shapes` to provide methods to support algorithms like these, which would make adding new `Shapes` to the system more difficult, we will instead implement an approach that places no new requirements on `Shapes`.

In the implementation here, we first generate random candidate points on the surfaces of translucent objects. These points are then either accepted or rejected based on a Poisson sphere test: if no other accepted points are within a given 3D distance of the candidate point, the point is accepted. This process continues until a few thousand candidate tests have been rejected in a row, suggesting that no more will be found that pass the Poisson sphere test. The approximation of using a 3D Poisson sphere criterion to approximate a 2D Poisson disk on the surface of a complex object generally works well, though for some geometric configurations it can be significantly inaccurate; Figure 16.10 shows some examples. (These cases tend to be ones where the semi-infinite slab approximation breaks down as well.) Exercise 16.12 at the end of the chapter discusses these issues further and suggests a few approaches to improve this part of the implementation. Figure 16.11 shows the collection of points computed by this algorithm for the dragon model.

This task is handled by the `SurfacePointsRenderer`, defined in the files `renderers/surfacepoints.h` and `renderers/surfacepoints.cpp`. This functionality is provided as a `Renderer` so that point sets can be precomputed and stored in a file for later use with the `DipoleSubsurfaceIntegrator`.

⟨*SurfacePointsRenderer Public Methods*⟩ ≡
```
SurfacePointsRenderer(float md, const Point &pc, float t,
                      const string &fn)
    : minDist(md), time(t), pCamera(pc), filename(fn) { }
```

DipoleSubsurfaceIntegrator 887
Point 63
Renderer 24
Shape 108

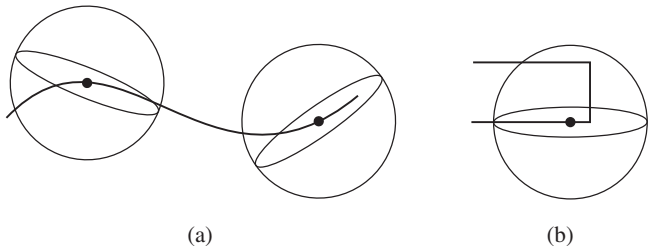


Figure 16.10: Cases Where the Poisson Sphere Criterion Works Well and Where It Fails. (a) For this relatively flat surface, distributing points with a Poisson sphere test gives a result similar to an actual Poisson disk test on the surface. (b) For this surface, with corners and a portion that folds over, the Poisson sphere criterion is too conservative, preventing points from being stored in some areas even though a Poisson disk test on the surface would include them.

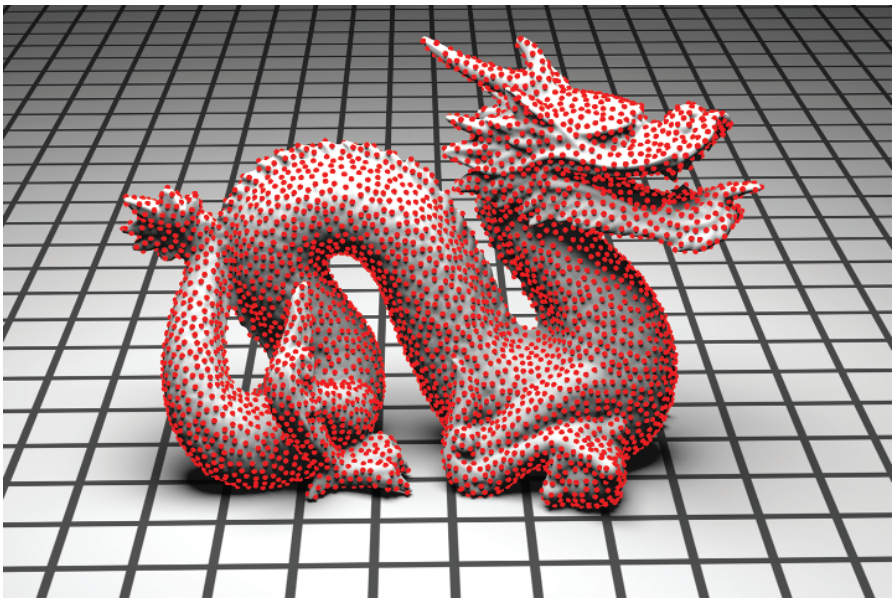


Figure 16.11: Sampled Points on the Dragon Model. This image shows a collection of points generated on the dragon model's surface using the point distribution algorithm implemented in this section.

⟨*SurfacePointsRenderer Private Data*⟩ ≡

```
float minDist, time;
Point pCamera;
string filename;
```

Point 63

To implement the Poisson sphere approach, we need to be able to generate an arbitrary number of random candidate points on translucent scene surfaces, and we need to

have an efficient way to reject the candidate points that don't fulfill the Poisson sphere requirement. To generate candidate points, we use a random sampling algorithm based on tracing rays through the scene and choosing a random scattered direction at each intersection point. This point sampling algorithm was introduced by Lehtinen et al. (2008) (though it wasn't applied to subsurface scattering). An octree is then used to store the accepted points, enabling efficient Poisson sphere tests for candidates. Because this funcionality is implemented in the context of a `Renderer` implementation, this work is done in the `Render()` method.

⟨*SurfacePointsRenderer Method Definitions*⟩ ≡
```
void SurfacePointsRenderer::Render(const Scene *scene) {
    ⟨Declare shared variables for Poisson point generation 890⟩
    ⟨Launch tasks to trace rays to find Poisson points⟩
    if (filename != "") {
        ⟨Write surface points to file⟩
    }
}
```

A number of common variables are needed by the tasks that will be computing these points. First, the tasks use a single shared `Octree` to store the accepted points and a few integer variables that record totals of the number of points that have been accepted, the number of points that have been rejected, and so forth. Once `maxFails` candidate points in a row have been rejected, the sampling algorithm terminates.

Due to the presence of these shared variables (which are periodically modified by the tasks), the tasks do require some synchronization to check and update their values; they use a single `RWMutex` for this task.

⟨*Declare shared variables for Poisson point generation*⟩ ≡ 890
```
BBox octBounds = scene->WorldBound();
octBounds.Expand(.001f * powf(octBounds.Volume(), 1.f/3.f));
Octree<SurfacePoint> pointOctree(octBounds);
⟨Create scene bounding sphere to catch rays that leave the scene 891⟩
int maxFails = 2000, repeatedFails = 0, maxRepeatedFails = 0;
int totalPathsTraced = 0, totalRaysTraced = 0, numPointsAdded = 0;
```

The accepted points are stored in an array of `SurfacePoints`. They are stored in a `SurfacePointsRenderer` member variable so that they persist after the `Render()` method exits.

⟨*SurfacePointsRenderer Private Data*⟩ +≡
```
vector<SurfacePoint> points;
```

⟨*SurfacePointsRenderer Declarations*⟩ ≡
```
struct SurfacePoint {
    SurfacePoint() { }
    SurfacePoint(const Point &pp, const Normal &nn, float a, float eps)
        : p(pp), n(nn), area(a), rayEpsilon(eps) { }
    ⟨SurfacePoint Data 891⟩
};
```

BBox 70
BBox::Expand() 72
BBox::Volume() 72
Normal 65
Octree 1023
Point 63
Renderer 24
RWMutex 1039
Scene 22
Scene::WorldBound() 24
SurfacePoint 890

⟨*SurfacePoint Data*⟩ ≡ 890
```
Point p;
Normal n;
float area, rayEpsilon;
```

The implementation uses a `GeometricPrimitive` that holds a sphere that surrounds the entire scene. For scenes that aren't closed, the point distribution algorithm needs an object like this that "catches" any rays that would otherwise leave the scene so that they can be redirected back into the scene.

⟨*Create scene bounding sphere to catch rays that leave the scene*⟩ ≡ 890, 962
```
Point sceneCenter;
float sceneRadius;
scene->WorldBound().BoundingSphere(&sceneCenter, &sceneRadius);
Transform ObjectToWorld(Translate(sceneCenter - Point(0,0,0)));
Transform WorldToObject(Inverse(ObjectToWorld));
Reference<Shape> sph = new Sphere(&ObjectToWorld, &WorldToObject,
    true, sceneRadius, -sceneRadius, sceneRadius, 360.f);
Reference<Material> nullMaterial = Reference<Material>(NULL);
GeometricPrimitive sphere(sph, nullMaterial, NULL);
```

`Tasks` are launched by the fragment ⟨*Launch tasks to trace rays to find Poisson points*⟩ (not included here) to generate the Poisson sphere points in parallel. The `SurfacePointTask::Run()` method does the computation. The computation in this method goes through three phases in turn:

1. Random ray paths are followed through the scene; at each intersection point, if the surface is translucent, then a candidate point is stored in local memory owned by the task. For all surfaces, a uniformly sampled direction is then used for the outgoing direction of a new ray starting at the intersection point. Note that this step requires just read-only access to the scene description.
2. After a few thousand ray paths have been followed and candidate points have been stored, the task acquires a reader lock on the reader–writer mutex that protects the octree. The task then checks all of the candidate points to see if there is an accepted point already in the octree that's too close to the candidate point, violating the Poisson sphere condition.
3. For the candidate points that pass the first test, the task then acquires a writer lock of the mutex, ensuring that no other thread will either read from or write to the octree during this time. The remaining points are checked again with the contents of the octree and the valid ones are added to it.

This process repeats until a large number of candidate points have been rejected in a row; at this point, the tasks exit under the assumption that no more valid points will be found.

This process was designed to minimize the amount of time that tasks held the writer lock, thus excluding all of other tasks from accessing the octree at all. Note that the second phase of the approach described above could be eliminated without affecting the correctness of the algorithm, though this would mean that tasks held the writer lock for

BBox::BoundingSphere() 74
GeometricPrimitive 188
Material 483
Normal 65
Point 63
Reference 1011
Scene::WorldBound() 24
Shape 108
Sphere 115
SurfacePointTask::Run() 892
Task 1041
Transform 76
Translate() 79

longer; if they can instead reject as many points as possible while using only read access, then they will spend less time holding the writer lock, and other tasks can potentially simultaneously test their candidate points using read access as well.

It is important to retest the points that passed the Poisson sphere test in the second step while actually holding the writer lock in the third step. Not only does the first test not compare the points with the other candidate points generated by the same task, but it's possible that another task may have acquired the writer lock and updated the tree between the time that the task does the first point tests with the read-only octree and the time that the task then adds the points in the read-write octree.

⟨*SurfacePointsRenderer Method Definitions*⟩ +≡
```
void SurfacePointTask::Run() {
    ⟨Declare common variables for SurfacePointTask::Run() 892⟩
    while (true) {
        int pathsTraced, raysTraced = 0;
        for (pathsTraced = 0; pathsTraced < 20000; ++pathsTraced) {
            ⟨Follow ray path and attempt to deposit candidate sample points 893⟩
        }
        ⟨Make first pass through candidate points with reader lock 894⟩
        ⟨Make second pass through points with writer lock and update octree 895⟩
        ⟨Stop following paths if not finding new points 896⟩
        candidates.erase(candidates.begin(), candidates.end());
    }
}
```

A few variables are initialized at the start of the task. The random number generator is seeded based on the number of the task currently running, ensuring that different tasks use different sequences of pseudo-random numbers. The `candidates` array stores the candidate points generated by the ray-tracing algorithm.

⟨*Declare common variables for* `SurfacePointTask::Run()`⟩ ≡ 892
```
RNG rng(37 * taskNum);
MemoryArena arena;
vector<SurfacePoint> candidates;
```

The candidate point distribution algorithm needs a point in the scene from which to start the ray paths. Paths from that point should be able to reach the surfaces of all translucent objects in the scene. In particular, this means that the point should not be inside a wall or other closed object in the scene. Here, the paths start from the camera position, which is passed to the `SurfacePointTask` constructor. This works well for most scenes, though there are reasonable scenes for which it doesn't work well; consider a translucent object seen through a glass window—paths starting from the camera will never go through the window, since the path generation algorithm here samples new directions in the same hemisphere as the incident ray.

Another alternative would be to allow multiple starting points to be provided, or to randomly select starting points from inside the overall scene bounding box. However,

MemoryArena 1015
RNG 1003
SurfacePoint 890

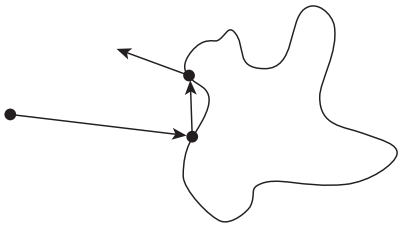


Figure 16.12: The Need for a Scene Bounding Sphere. If the scene is only comprised of a single triangle mesh, for example, and if all ray paths start from a single point (filled circle on left), then we need a sphere that bounds the scene for rays to intersect so that they can find paths to the backside of the object with respect to the path starting point.

selecting random points in the scene bounds would be inefficient in an extremely complex model like an office building where the camera was just looking at the contents of a single room; few of the paths would pass through the visible scene as seen by the camera.

The first few ray intersections along this path are ignored, as we'd like to generate a reasonably uniform distribution of random candidate points. (It's not critical that the distribution of candidate points be random, only that the ray paths reach all of the translucent objects in the scene.) For all rays that do hit a translucent object (i.e., one with a non-`NULL` BSSRDF), the point and some of its local geometric information are stored in the `candidates` array.

⟨*Follow ray path and attempt to deposit candidate sample points*⟩ ≡ 892

```
Vector dir = UniformSampleSphere(rng.RandomFloat(), rng.RandomFloat());
Ray ray(origin, dir, 0.f, INFINITY, time);
while (ray.depth < 30) {
    ⟨Find ray intersection with scene geometry or bounding sphere 894⟩
    ⟨Store candidate sample point at ray intersection if appropriate 894⟩
    ⟨Generate random ray from intersection point 894⟩
}
arena.FreeAll();
```

If the current ray doesn't hit any scene geometry, we test for intersection with the sphere that bounds the scene. Figure 16.12 shows why this sphere is needed; consider a scene composed of just a single translucent triangle mesh. With ray paths starting from a single point, multiple bounces between points on the mesh will not in general be able to reach some parts of the model's surface.

After a hit is found, the surface normal at the hit point is reoriented if needed so that it faces on the side of the surface that the ray arrived from.

`INFINITY` 1002
`MemoryArena::FreeAll()` 1017
`Ray` 66
`Ray::depth` 67
`RNG::RandomFloat()` 1003
`UniformSampleSphere()` 664
`Vector` 57

⟨*Find ray intersection with scene geometry or bounding sphere*⟩ ≡ 893
```
++raysTraced;
Intersection isect;
bool hitOnSphere = false;
if (!scene->Intersect(ray, &isect)) {
    if (!sphere.Intersect(ray, &isect))
        break;
    hitOnSphere = true;
}
DifferentialGeometry &hitGeometry = isect.dg;
hitGeometry.nn = Faceforward(hitGeometry.nn, -ray.d);
```

For points on translucent surfaces that have undergone a few bounces in the path, a `SurfacePoint` is added to the candidate array. The area on the surface that the point's sample represents is approximated with the area of the circle from the Poisson criterion. This approximation is accurate for reasonably flat surfaces and works well in practice.

⟨*Store candidate sample point at ray intersection if appropriate*⟩ ≡ 893
```
if (!hitOnSphere && ray.depth >= 3 &&
    isect.GetBSSRDF(RayDifferential(ray), arena) != NULL) {
    float area = M_PI * minSampleDist * minSampleDist;
    candidates.push_back(SurfacePoint(hitGeometry.p, hitGeometry.nn,
                                      area, isect.rayEpsilon));
}
```

At the intersection point, a uniform direction is sampled in the hemisphere around the surface normal at the point so that the new ray leaves the surface on the same side from which the previous one arrived. In general, we don't want the rays to get caught inside closed objects in the scene (like walls), though as noted above this choice means that we may miss sampling some parts of the scene that are visible through transparent objects.

⟨*Generate random ray from intersection point*⟩ ≡ 893
```
Vector dir = UniformSampleSphere(rng.RandomFloat(), rng.RandomFloat());
dir = Faceforward(dir, hitGeometry.nn);
ray = Ray(hitGeometry.p, dir, ray, isect.rayEpsilon);
```

After a number of paths have been followed and, presumably, candidate points generated, the second phase starts, with the read lock held. We'd like to reject any points that we can while only holding a read lock, thus reducing the time a write lock is held later.

⟨*Make first pass through candidate points with reader lock*⟩ ≡ 892
```
vector<bool> candidateRejected;
RWMutexLock lock(mutex, READ);
for (uint32_t i = 0; i < candidates.size(); ++i) {
    PoissonCheck check(minSampleDist, candidates[i].p);
    octree.Lookup(candidates[i].p, check);
    candidateRejected.push_back(check.failed);
}
```

The `PoissonCheck` helper class is called by the octree for all of the points that it stores that could be within the minimum distance of the point passed to its `Lookup()` method. If it

DifferentialGeometry 102
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Faceforward() 66
GeometricPrimitive::
Intersect() 188
Intersection 186
Intersection::
GetBSSRDF() 484
Intersection::rayEpsilon 186
M_PI 1002
Octree::Lookup() 1027
PoissonCheck 895
Ray 66
RayDifferential 69
RNG::RandomFloat() 1003
RWMutexLock 1039
Scene::Intersect() 23
SurfacePoint 890
UniformSampleSphere() 664
Vector 57

finds any point closer than the maximum distance, it updates `failed` and returns `false`, indicating to the octree that it should stop any further traversal.

⟨*SurfacePointsRenderer Local Declarations*⟩ ≡
```
struct PoissonCheck {
    PoissonCheck(float md, const Point &pt)
        { maxDist2 = md * md; failed = false; p = pt; }
    float maxDist2;
    bool failed;
    Point p;
    bool operator()(const SurfacePoint &sp) {
        if (DistanceSquared(sp.p, p) < maxDist2) {
            failed = true; return false;
        }
        return true;
    }
};
```

The task now takes a write lock; it can now update the shared variables and octree knowing that no other task will be reading or writing any of them concurrently. The `repeatedFails` variable records how many candidate points in a row have been rejected. Another task may have hit the limit, in which case all of the other tasks exit the next time they acquire the write lock and discover that candidate generation should stop.

⟨*Make second pass through points with writer lock and update octree*⟩ ≡ 892
```
lock.UpgradeToWrite();
if (repeatedFails >= maxFails)
    return;
totalPathsTraced += pathsTraced;
totalRaysTraced += raysTraced;
for (uint32_t i = 0; i < candidates.size(); ++i) {
    if (candidateRejected[i]) {
        ⟨Update for rejected candidate point 895⟩
    }
    else {
        ⟨Recheck candidate point and possibly add to octree 896⟩
    }
}
```

For each rejected point, `repeatedFails` is incremented; when it hits the maximum, the task exits.

DistanceSquared() 65
IrradiancePoint::p 898
Point 63
PoissonCheck 895
RWMutexLock::
 UpgradeToWrite() 1039
SurfacePoint 890

⟨*Update for rejected candidate point*⟩ ≡ 895, 896
```
++repeatedFails;
if (repeatedFails >= maxFails)
    return;
```

For points that passed the first check, we need to verify that they still pass the Poisson sphere test. As mentioned previously, the effect of updates to the octree from other tasks or from other points in this task's set of candidates means that the candidate point may

not in fact still be valid. If it again passes, the octree is updated and the point is added to surfacePoints, which is a member variable of the SurfacePointsRenderer class passed into the tasks by reference.

⟨*Recheck candidate point and possibly add to octree*⟩ ≡ 895
```
SurfacePoint &sp = candidates[i];
PoissonCheck check(minSampleDist, sp.p);
octree.Lookup(sp.p, check);
if (check.failed) {
    ⟨Update for rejected candidate point 895⟩
}
else {
    ++numPointsAdded;
    repeatedFails = 0;
    Vector delta(minSampleDist, minSampleDist, minSampleDist);
    octree.Add(sp, BBox(sp.p-delta, sp.p+delta));
    surfacePoints.push_back(sp);
}
```

Finally, it's possible that many paths will be traced but no translucent objects are found. In this case, a warning is printed and the tasks exit.

⟨*Stop following paths if not finding new points*⟩ ≡ 892
```
if (totalPathsTraced > 50000 && numPointsAdded == 0) {
    Warning("There don't seem to be any objects with BSSRDFs "
            "in this scene.  Giving up.");
    return;
}
```

After the tasks have exited, the points, their surface normals, their corresponding areas, and ray epsilon values are written to a file if a filename was provided. This is handled by the ⟨*Write surface points to file*⟩ fragment, which is straightforward and thus not included here.

For the convenience of the DipoleSubsurfaceIntegrator, a short utility function can be called to compute a surface point set for a scene without the caller needing to explicitly create a SurfacePointsRenderer.

⟨*SurfacePointsRenderer Method Definitions*⟩ +≡
```
void FindPoissonPointDistribution(const Point &pCamera, float time,
        float minDist, const Scene *scene, vector<SurfacePoint> *points) {
    SurfacePointsRenderer sp(minDist, pCamera, time, "");
    sp.Render(scene);
    points->swap(sp.points);
}
```

BBox 70
DipoleSubsurfaceIntegrator 887
Octree::Add() 1024
Octree::Lookup() 1027
Point 63
PoissonCheck 895
Scene 22
SurfacePoint 890
SurfacePointsRenderer 888
Vector 57

16.5.2 BUILDING THE SAMPLE POINT OCTREE

We can now turn to the implementation of the `Preprocess()` method of the `DipoleSubsurfaceIntegrator`. It starts by getting a collection of `SurfacePoints` on the surfaces of translucent objects, either loading them from a file generated by the `SurfacePointsRenderer` or generating them here. Incident irradiance is then computed at each point, and then an octree that encodes a hierarchical clustering of the points is created.

⟨*DipoleSubsurfaceIntegrator Method Definitions*⟩ ≡
```
void DipoleSubsurfaceIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    if (scene->lights.size() == 0) return;
    vector<SurfacePoint> pts;
    ⟨Get SurfacePoints for translucent objects in scene 897⟩
    ⟨Compute irradiance values at sample points 897⟩
    ⟨Create octree of clustered irradiance samples 898⟩
}
```

⟨*Get* `SurfacePoints` *for translucent objects in scene*⟩ ≡ 897
```
if (filename != "") {
    ⟨Initialize SurfacePoints from file⟩
}
if (pts.size() == 0) {
    Point pCamera = camera->CameraToWorld(camera->shutterOpen,
                                          Point(0, 0, 0));
    FindPoissonPointDistribution(pCamera, camera->shutterOpen,
                                 minSampleDist, scene, &pts);
}
```

For each `SurfacePoint`, the incident irradiance from the light sources in the scene is computed. An `IrradiancePoint` is then created for each point.

⟨*Compute irradiance values at sample points*⟩ ≡ 897
```
RNG rng;
MemoryArena arena;
for (uint32_t i = 0; i < pts.size(); ++i) {
    SurfacePoint &sp = pts[i];
    Spectrum E(0.f);
    for (uint32_t j = 0; j < scene->lights.size(); ++j) {
        ⟨Add irradiance from light at point⟩
    }
    irradiancePoints.push_back(IrradiancePoint(sp, E));
    arena.FreeAll();
}
```

⟨*DipoleSubsurfaceIntegrator Private Data*⟩ +≡
```
vector<IrradiancePoint> irradiancePoints;
```

Camera 302
Camera::CameraToWorld 302
Camera::shutterOpen 302
FindPoissonPoint Distribution() 896
IrradiancePoint 898
MemoryArena 1015
MemoryArena::FreeAll() 1017
Point 63
Renderer 24
RNG 1003
Scene 22
Scene::lights 23
Spectrum 263
SurfacePoint 890
SurfacePointsRenderer 888

⟨*DipoleSubsurfaceIntegrator Helper Declarations*⟩ ≡
```
struct IrradiancePoint {
    IrradiancePoint(const SurfacePoint &sp, const Spectrum &ee)
        : p(sp.p), n(sp.n), E(ee), area(sp.area),
          rayEpsilon(sp.rayEpsilon) { }
    Point p;
    Normal n;
    Spectrum E;
    float area, rayEpsilon;
};
```

The implementation of the fragment ⟨*Add irradiance from light at point*⟩ is straightforward and not included here; it computes the contribution of direct lighting to irradiance, ignoring indirect illumination, taking a number of samples for each light to estimate the integral from Equation (5.3). This computation is currently done in a single thread of execution; it is usually fast enough that breaking it into tasks is of limited value. For many applications, it is desirable to instead compute irradiance including indirect illumination. If the indirect illumination computation is more computationally intensive, parallelization of this step may be appropriate.

After the irradiance values for the `IrradiancePoints` are computed, the octree is created. The `Preprocess()` method allocates the root node of the tree, computes the bounding box of all of the points, and inserts all of the points into the octree in turn. The root node starts out as an empty leaf node but progressively refines itself to have children nodes (which in turn refine themselves) as more points are added to it.

⟨*Create octree of clustered irradiance samples*⟩ ≡ 897
```
octree = octreeArena.Alloc<SubsurfaceOctreeNode>();
for (uint32_t i = 0; i < irradiancePoints.size(); ++i)
    octreeBounds = Union(octreeBounds, irradiancePoints[i].p);
for (uint32_t i = 0; i < irradiancePoints.size(); ++i)
    octree->Insert(octreeBounds, &irradiancePoints[i], octreeArena);
octree->InitHierarchy();
```

⟨*DipoleSubsurfaceIntegrator Private Data*⟩ +≡
```
BBox octreeBounds;
SubsurfaceOctreeNode *octree;
MemoryArena octreeArena;
```

The `SubsurfaceOctreeNode` class represents a node of the octree that stores the hierarchical clustered irradiance points.

⟨*DipoleSubsurfaceIntegrator Local Declarations*⟩ ≡
```
struct SubsurfaceOctreeNode {
    ⟨SubsurfaceOctreeNode Methods 899⟩
    ⟨SubsurfaceOctreeNode Public Data 899⟩
};
```

BBox 70
DipoleSubsurfaceIntegrator::
irradiancePoints 897
DipoleSubsurfaceIntegrator::
octree 898
DipoleSubsurfaceIntegrator::
octreeArena 898
IrradiancePoint 898
MemoryArena 1015
MemoryArena::Alloc() 1016
Normal 65
Point 63
Spectrum 263
SubsurfaceOctreeNode 898
SubsurfaceOctreeNode::
InitHierarchy() 901
SubsurfaceOctreeNode::
Insert() 899
SurfacePoint 890
Union() 72

The leaf nodes of the octree hold pointers to up to eight `IrradiancePoints`. Interior nodes hold pointers to the eight potential children. This octree follows the conventions of the `Octree` template class in Section A.7 in terms of the ordering of the children. Each node of the octree also holds aggregate information about the `IrradiancePoints` in the tree beneath it, including the average position and irradiance of the points underneath it as well as the sum of their areas.

⟨*SubsurfaceOctreeNode Public Data*⟩ ≡ 898
```
Point p;
bool isLeaf;
Spectrum E;
float sumArea;
union {
    SubsurfaceOctreeNode *children[8];
    IrradiancePoint *ips[8];
};
```

⟨*SubsurfaceOctreeNode Methods*⟩ ≡ 898
```
SubsurfaceOctreeNode() {
    isLeaf = true;
    sumArea = 0.f;
    for (int i = 0; i < 8; ++i)
        ips[i] = NULL;
}
```

The `Insert()` method first computes the midpoint of the current node, which will be used by all code paths later in the method, and then determines what to do with the point being added. (It is a precondition that the point is inside the current node.) If the node is a leaf, then the pointer to the point is added to the points stored in the leaf; otherwise the `Insert()` method of the child node that the point overlaps will be called, progressing recursively until a leaf node is reached.

⟨*SubsurfaceOctreeNode Methods*⟩ +≡ 898
```
void Insert(const BBox &nodeBound, IrradiancePoint *ip,
            MemoryArena &arena) {
    Point pMid = .5f * nodeBound.pMin + .5f * nodeBound.pMax;
    if (isLeaf) {
        ⟨Add IrradiancePoint to leaf octree node 900⟩
    }
    ⟨Add IrradiancePoint ip to interior octree node 900⟩
}
```

Leaf nodes can only store eight `IrradiancePoints`; the pointers for them start out all `NULL` and are initialized in turn. If all of the pointers are in use when a new point is added, then the node converts itself to an interior node and distributes its `IrradiancePoints` to newly allocated children nodes.

BBox 70
BBox::pMax 71
BBox::pMin 71
IrradiancePoint 898
MemoryArena 1015
Octree 1023
Point 63
Spectrum 263
SubsurfaceOctreeNode 898
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899
SubsurfaceOctreeNode::sumArea 899

⟨*Add* `IrradiancePoint` *to leaf octree node*⟩ ≡ 899
```
for (int i = 0; i < 8; ++i) {
    if (!ips[i]) {
        ips[i] = ip;
        return;
    }
}
⟨Convert leaf node to interior node, redistribute points 900⟩
/* fall through to interior case to insert the new point... */
```

To convert a leaf to an interior node, the `IrradiancePoint` pointers first must be copied to an array separate from the one in the node. Recall that the pointers to the `IrradiancePoints` share a `union` with the pointers to the children; if the pointers weren't copied first, then they'd be clobbered as we allocated new children nodes and set their pointers in the code below. Even worse, the implementation would interpret `IrradiancePoint` pointers as `SubsurfaceOctreeNode` pointers, leading to major errors or crashes.

After the points are copied, the `children` pointers are set to `NULL` and the `IrradiancePoint` pointers are inserted into newly allocated children of this node.

⟨*Convert leaf node to interior node, redistribute points*⟩ ≡ 900
```
isLeaf = false;
IrradiancePoint *localIps[8];
for (int i = 0; i < 8; ++i) {
    localIps[i] = ips[i];
    children[i] = NULL;
}
for (int i = 0; i < 8; ++i)  {
    IrradiancePoint *ip = localIps[i];
    ⟨Add IrradiancePoint ip to interior octree node 900⟩
}
```

The following fragment is used twice; once in the `for` loop in the fragment immediately above here, and once in the definition of `SubsurfaceOctreeNode::Insert()`.

⟨*Add* `IrradiancePoint ip` *to interior octree node*⟩ ≡ 899, 900
```
int child = (ip->p.x > pMid.x ? 4 : 0) +
    (ip->p.y > pMid.y ? 2 : 0) + (ip->p.z > pMid.z ? 1 : 0);
if (!children[child])
    children[child] = arena.Alloc<SubsurfaceOctreeNode>();
BBox childBound = octreeChildBound(child, nodeBound, pMid);
children[child]->Insert(childBound, ip, arena);
```

Once all of the points have been added to the octree, `InitHierarchy()` is called. This method computes the aggregate values at each node: the average of the positions and irradiance values and the sum of the areas of the children.

BBox 70
IrradiancePoint 898
IrradiancePoint::p 898
MemoryArena::Alloc() 1016
octreeChildBound() 1026
SubsurfaceOctreeNode 898
SubsurfaceOctreeNode::children 899
SubsurfaceOctreeNode::Insert() 899
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899

⟨*SubsurfaceOctreeNode Methods*⟩ +≡ 898
```
void InitHierarchy() {
    if (isLeaf) {
        ⟨Init SubsurfaceOctreeNode leaf from IrradiancePoints 901⟩
    }
    else {
        ⟨Init interior SubsurfaceOctreeNode⟩
    }
}
```

At leaf nodes, the average irradiance and the total area that the points represent are both easily computed from the individual `IrradiancePoints`. The representative position for the samples is computed as a luminance-weighted average of the positions of the points; the idea is that points with relatively high irradiance values should have more weight in computing the average position used by the forthcoming hierarchical integration algorithm.

⟨*Init* `SubsurfaceOctreeNode` *leaf from* `IrradiancePoints`⟩ ≡ 901
```
float sumWt = 0.f;
uint32_t i;
for (i = 0; i < 8; ++i) {
    if (!ips[i]) break;
    float wt = ips[i]->E.y();
    E += ips[i]->E;
    p += wt * ips[i]->p;
    sumWt += wt;
    sumArea += ips[i]->area;
}
if (sumWt > 0.f) p /= sumWt;
E /= i;
```

At interior nodes, the ⟨*Init interior* `SubsurfaceOctreeNode`⟩ fragment first computes aggregate values for the children nodes via recursive calls to `InitHierarchy()`. Then, it computes aggregate values directly from the children's values. The computation otherwise follows the form of the computation in the fragment ⟨*Init* `SubsurfaceOctreeNode` *leaf from* `IrradiancePoints`⟩ and so won't be included here.

DipoleSubsurfaceIntegrator 887
IrradiancePoint 898
IrradiancePoint::area 898
IrradiancePoint::E 898
IrradiancePoint::p 898
Spectrum::y() 273
SubsurfaceOctreeNode::E 899
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899
SubsurfaceOctreeNode::p 899
SubsurfaceOctreeNode::sumArea 899

16.5.3 THE DIPOLE DIFFUSION APPROXIMATION

We will give an overview of the diffusion approximation of the equation of transfer as well as the dipole method for solving it; together these lead to the efficient approach used for subsurface scattering in the `DipoleSubsurfaceIntegrator`. The full derivations of each of these are lengthy and complex; as such, we won't include them in the text here. See Chapter 5 of Donner's Ph.D. dissertation (2006) for a full derivation of both. The "Further Reading" section also has pointers to additional information on these topics.

A number of important ideas are used in the process of transforming the fully general equation of transfer to the diffusion equation, which can be approximately solved for subsurface scattering. The first is the *principle of similarity*, which says that for an

anisotropically scattering medium with a high albedo, the medium can instead be modeled as having an isotropic phase function with appropriately modified scattering and extinction coefficients. Light transport solutions computed based on the modified coefficients correspond well to those with the original coefficients and phase function, while allowing simplifications due to the assumption of isotropic scattering.

The principle of similarity is based on the observation that after many scattering events the distribution of light in media with high albedos becomes more and more uniformly directionally distributed regardless of the original illumination distribution or the anisotropy of the phase function. One way to see how this happens is to consider an expression derived by Yanovitskij (1997); it describes isotropization due to multiple scattering events from the Henyey–Greenstein phase function. He showed that the distribution of light that has been scattered n times is given by

$$p_n(\omega \to \omega') = \frac{1 - g^{2n}}{4\pi(1 + g^{2n} - 2g|g^{n-1}|\,(-\omega \cdot \omega')^{3/2})}.$$

As n grows large, this converges to the isotropic phase function, $1/4\pi$. Figure 16.13 plots this function for a few values of n. When coupled with the observation from the start of this section about how much of the light energy remains after tens or even hundreds of scattering events in high-albedo materials, one can see intuitively why it is reasonable to work with an isotropic phase function approximation for high-albedo media.

The *reduced scattering coefficient* is defined as $\sigma_s' = (1 - g)\sigma_s$, where g is the anisotropy parameter, and the *reduced extinction coefficient* is $\sigma_t' = \sigma_a + \sigma_s'$. These new coefficients account for the effect of using the isotropic phase function approximation. To understand the idea they embody, consider a strongly forward-scattering phase function, where $g \to 1$. With the actual phase function, at each scattering event the light will mostly continue in the same direction to its next scattering event. In this case, the value of the reduced scattering coefficient σ_s' is much smaller than σ_s, which means that light travels a larger distance in the medium before scattering; the medium is effectively approximated as being thinner, allowing light to travel further, having the same effect as a highly forward-scattering phase function.

Conversely, consider the case of $g \to -1$. In this case, at a scattering event the light will tend to scatter back in the direction it came from. But then the next time it scatters after that, it will generally reverse course again; it bounces back and forth without making very much forward progress. In this case, the reduced scattering coefficient is larger than the original scattering coefficient, indicating greater probability of scattering interaction. In other words, the medium is treated as being thicker than it actually is, which approximates the effect of light having relatively more trouble making forward progress. Figure 16.14 illustrates these ideas, showing representative paths of scattering interactions in highly forward-scattering and highly backward-scattering media.

The diffusion equation can be derived from the equation of transfer; it provides a solution to the equation of transfer for the case of homogeneous, optically thick, highly scattering materials (i.e., those with relatively large albedos). For the application to subsurface scattering in `pbrt`, it can be derived by writing the equation of transfer using the reduced scattering and extinction coefficients and an isotropic phase function. Then, the terms that account for single scattering and light transmitted directly through the me-

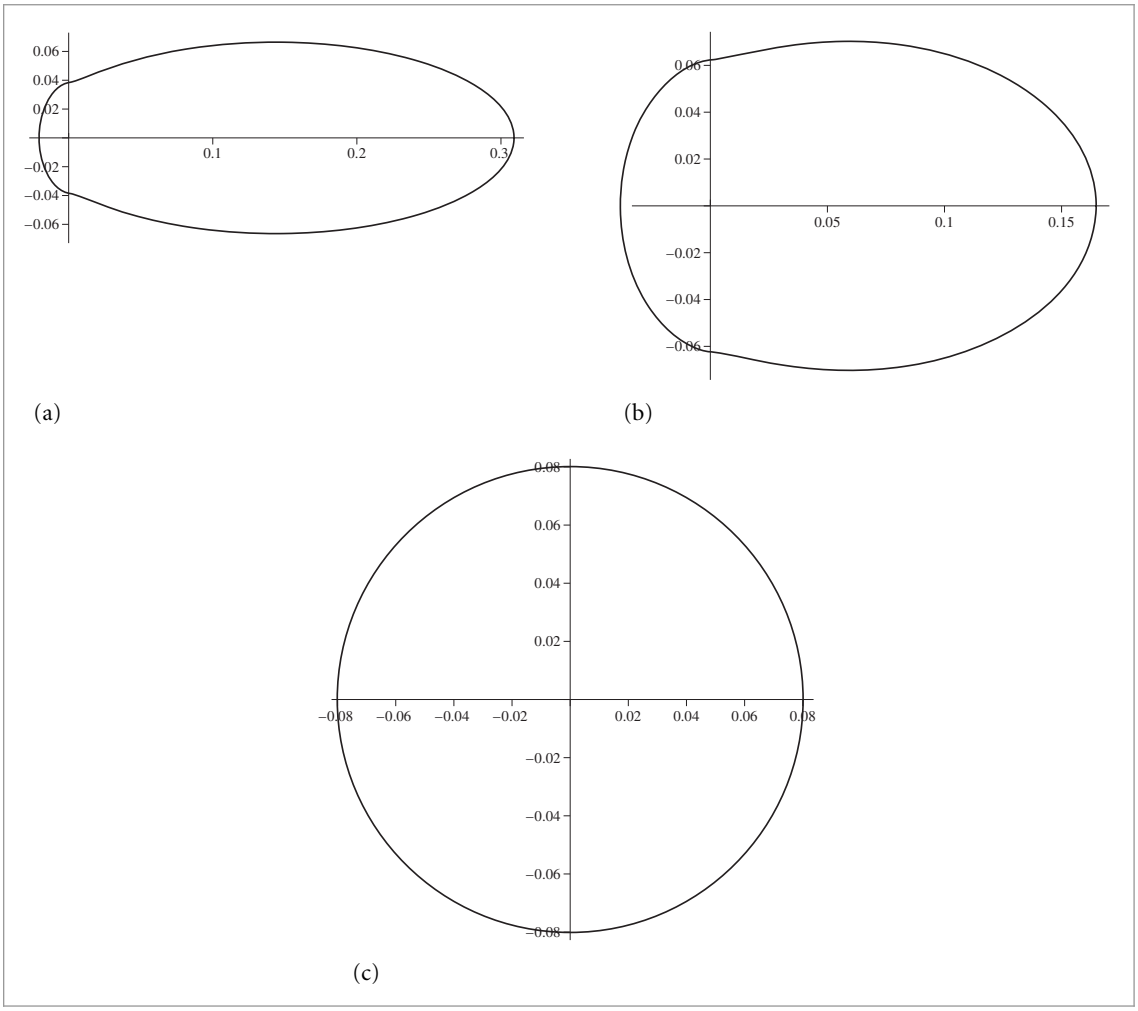


Figure 16.13: Light Distribution After Many Scattering Events. (a) Directional distribution of a single incident ray of light after 10 scattering events in a highly anisotropic medium with $g = 0.9$, (b) 100 scattering events, (c) 1000 scattering events. The distribution becomes increasingly isotropic, even though it was initially very anisotropic.

dium are removed, so that only multiple scattering is considered. (Those two terms are easily handled separately.)

The resulting spatially varying radiance function $L(\mathrm{p}, \omega)$ is expressed with four terms in the spherical harmonic basis (Section 17.2), essentially as a constant term plus linear functions of x, y, and z. The result is the diffusion equation,

$$\nabla \cdot \left(\frac{\nabla \phi(\mathrm{p})}{\sigma_{\mathrm{t}}'} \right) - 3\sigma_{\mathrm{a}} \phi(\mathrm{p}) = 0$$

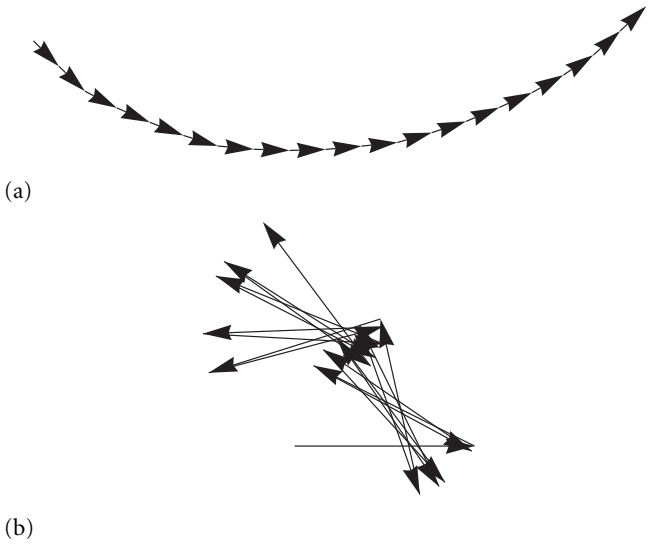


Figure 16.14: Representative Light Paths for Highly Anisotropic Scattering Media. (a) Forward-scattering medium, with $g = 0.95$. Light generally scatters in the same direction it was originally traveling. (b) Backward-scattering medium, with $g = -0.95$. Light frequently bounces back and forth, making relatively little forward progress with respect to its original direction.

where ϕ represents *fluence*,

$$\phi(\mathrm{p}) = \int_{S^2} L_\mathrm{i}(\mathrm{p}, \omega)\, \mathrm{d}\omega.$$

The diffusion equation can be solved in certain conditions. For example, the fluence at a point $\mathrm{p_i}$ from a point light source source at point p with power Φ in an infinite medium has an analytic solution,

$$\phi(\mathrm{p}, \mathrm{p_i}) = \frac{\Phi}{4\pi D} \frac{\mathrm{e}^{-\sigma_\mathrm{tr}\|\mathrm{p}-\mathrm{p_i}\|}}{\|\mathrm{p} - \mathrm{p_i}\|} \qquad [16.3]$$

where $\sigma_\mathrm{tr} = \sqrt{3\sigma_\mathrm{t}'\sigma_\mathrm{a}}$, and D is the diffusion coefficient $D = 1/(3\sigma_\mathrm{t}')$.

For subsurface scattering, it's useful to consider the case of a semi-infinite slab: the scattering medium is all space below the $z = 0$ plane. To solve this diffusion equation in this case, a suitable boundary condition must be found. In particular, we'd like a boundary condition that lets us compute fluence on the boundary of the medium; it's not important to be able to compute the scattering at points inside of it, since we're only interested here in computing the light leaving it.

In the semi-infinite slab setting, there's no incident radiance at the top of the slab due to radiance from scattering within the slab. Thus, the incident radiance at the boundary in the hemisphere around the $(0, 0, -1)$ direction is equal to the incident radiance from scattering in the medium arriving at the boundary reflected by the Fresnel effect back

down into the medium. This boundary condition can be approximated as $\phi(2AD) = 0$, where

$$A = \frac{1 + F_{\mathrm{dr}}(\eta)}{1 - F_{\mathrm{dr}}(\eta)}$$

and F_{dr} is the diffuse Fresnel reflectance, given by

$$F_{\mathrm{dr}}(\eta) = \int_{\mathcal{H}^2(\mathbf{n})} F_{\mathrm{r}}(\eta, \omega_{\mathrm{i}})\, |\omega_{\mathrm{i}} \cdot \mathbf{n}|\, \mathrm{d}\omega_{\mathrm{i}}. \tag{16.4}$$

This boundary condition says that fluence goes to zero at a distance $2AD$ above the slab's boundary; it represents the extrapoloation of the fall-off of fluence inside the medium as the boundary is approached and thus leads to a reasonable solution for points on the actual boundary of the slab. Note that in general a point outside the medium at that distance would expect to have nonzero fluence due to radiance leaving the medium; however, our goal here is to compute a good solution at the boundary of the medium.

A short utility function computes the diffuse Fresnel reflectance using a closed-form rational approximation to Equation (16.4).

⟨*Reflection Declarations*⟩ +≡

```
inline float Fdr(float eta) {
    if (eta >= 1)
        return -1.4399f / (eta*eta) + 0.7099f / eta + 0.6681f +
            0.0636f * eta;
    else
        return -0.4399f + .7099f / eta - .3319f / (eta * eta) +
            .0636f / (eta*eta*eta);
}
```

One way to generate a fluence distribution that matches the boundary condition $\phi(2AD) = 0$ is to use the analytic solution for a point source, Equation (16.3), to compute the fluence for a pair of light sources, one with positive flux and one with negative flux. If these two light sources are placed appropriately, they will cancel out along the plane $z = 2AD$, fulfilling the boundary condition.

If light enters the medium at the point $\mathrm{p_i} = (x_{\mathrm{i}}, y_{\mathrm{i}}, 0)$ on the boundary, the *dipole approximation* uses two such sources to fulfill the boundary condition, with the positive point inside the medium at the point $(x_{\mathrm{i}}, y_{\mathrm{i}}, z^+)$ and the negative point above the medium at the point $(x_{\mathrm{i}}, y_{\mathrm{i}}, z^-)$ with $z^- = 2AD + z^+$. Thus, the fluence leaving the medium at a particular point $\mathrm{p_o} = (x_{\mathrm{o}}, y_{\mathrm{o}}, 0)$ is given by subtracting the contributions of the negative source from the contribution of the positive source:

$$\phi(\mathrm{p_i}, \mathrm{p_o}) = \frac{\Phi}{4\pi D}\left(\frac{\mathrm{e}^{-\sigma_{\mathrm{tr}} d^+}}{d^+} - \frac{\mathrm{e}^{-\sigma_{\mathrm{tr}} d^-}}{d^-}\right), \tag{16.5}$$

with the distance from the point to the sources given by

$$d^+ = \|(x_{\mathrm{i}}, y_{\mathrm{i}}, z^+) - \mathrm{p_o}\|, \text{ and } d^- = \|(x_{\mathrm{i}}, y_{\mathrm{i}}, z^-) - \mathrm{p_o}\|.$$

See Figure 16.15.

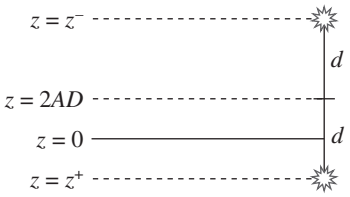


Figure 16.15: Basic Setting for the Dipole Approximation to the Solution to the Diffusion Equation. A light source with positive flux is placed at a position $z = z^+$ inside the medium, beneath the point where incident illumination arrives, and a second light with an equal amount of negative flux is placed at $z = z^-$ above the medium. These sources are placed so that the flux for the two of them cancels out at a height $z = 2AD$ above the boundary $z = 0$, fulfilling the desired boundary condition. The fluence that results from subtracting the closed-form expression of the fluence distributions for each of them (Equation (16.3)) at the medium's boundary $z = 0$ is a reasonable approximation of the fluence at the boundary due to subsurface scattering. If we use this approximation to compute the fluence at a point $\mathrm{p_o}$ on the boundary, we can find how much light is exiting the medium due to light entering at $\mathrm{p_i}$.

To compute the diffuse subsurface reflectance, we'd like to compute the radiant exitance, which is found from the fluence by taking the gradient of the fluence function and computing the dot product with the normal $\mathbf{n}$,

$$M_\mathrm{o}(\mathrm{p_o}, \mathbf{n}) = -D(\mathbf{n} \cdot \nabla\phi)(\mathrm{p_o}).$$

Equivalently, and more conveniently, we can take the derivative with respect to z,

$$M_\mathrm{o}(\mathrm{p_o}, \mathbf{n}) = -D\frac{\mathrm{d}}{\mathrm{d}z}\phi(\mathrm{p_o}).$$

The differential diffuse subsurface reflectance R_d due to illumination at a point $\mathrm{p_i}$ is given in turn by taking the differential $\mathrm{d}M_\mathrm{o}(\mathrm{p_o}, \mathbf{n})/\mathrm{d}\Phi(\mathrm{p_i})$.

Putting this all together and working through the derivative of Equation (16.5), we finally have the dipole approximation to diffuse subsurface reflectance, which we'll use for rendering in the implementation below.

$$R_\mathrm{d}(\mathrm{p_i}, \mathrm{p_o}) = \frac{1}{4\pi}\left(\frac{z^+(d^+\sigma_\mathrm{tr} + 1)\mathrm{e}^{-\sigma_\mathrm{tr}d^+}}{(d^+)^3} - \frac{z^-(d^-\sigma_\mathrm{tr} + 1)\mathrm{e}^{-\sigma_\mathrm{tr}d^-}}{(d^-)^3}\right). \quad [16.6]$$

Figure 16.16 is a graph of this function. Diffuse subsurface reflectance R_d is the analog of the reflectance of BSDFs ρ_hd and ρ_hh introduced in Section 8.1.1, in that it represents directionally averaged scattering.

`DiffusionReflectance` is a small utility class that computes the value of R_d using Equation (16.6). Rather than encapsulating this computation in a utility function, using a class makes it easier to precompute a number of common terms that only depend on the scattering properties of the medium. Their values can then be reused many times as R_d is evaluated for many different points.

`DiffusionReflectance` 907

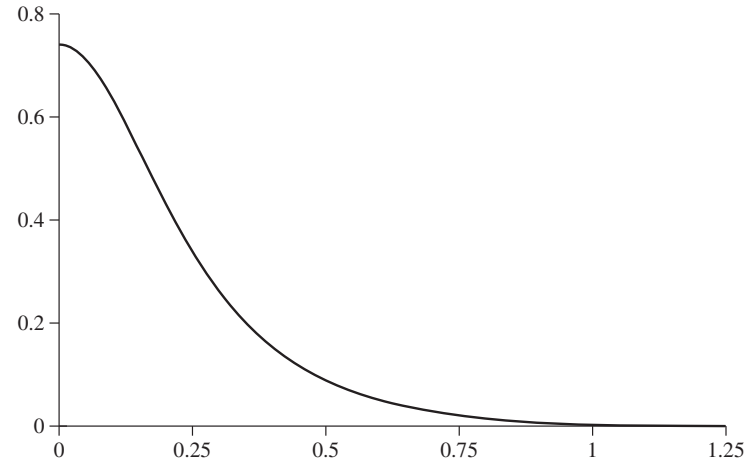


Figure 16.16: Graph of the R_d Function from Equation (16.6), Using Measured Scattering Properties of Milk. The value of R_d falls off rapidly as a function of distance d between the points p_i and p_o.

⟨*DipoleSubsurfaceIntegrator Local Declarations*⟩ +≡
```
struct DiffusionReflectance {
    ⟨DiffusionReflectance Public Methods 907⟩
    ⟨DiffusionReflectance Data 907⟩
};
```

The constructor precomputes the terms that are constant for all evaluations of R_d. Following Farrell et al. (1992), the positive source is placed at depth $z^+ = 1/\sigma_t'$ in the medium; this corresponds to the *mean free path*, the distance at which the first scattering event due to incident illumination at the top of the medium is expected to occur.

⟨*DiffusionReflectance Public Methods*⟩ ≡ 907
```
DiffusionReflectance(const Spectrum &sigma_a, const Spectrum &sigmap_s,
                     float eta) {
    A = (1.f + Fdr(eta)) / (1.f - Fdr(eta));
    sigmap_t = sigma_a + sigmap_s;
    sigma_tr = Sqrt(3.f * sigma_a * sigmap_t);
    alphap = sigmap_s / sigmap_t;
    zpos = Spectrum(1.f) / sigmap_t;
    zneg = zpos * (1.f + (4.f/3.f) * A);
}
```

DiffusionReflectance 907
Fdr() 905
Spectrum 263
Spectrum::Sqrt() 265

⟨*DiffusionReflectance Data*⟩ ≡ 907
```
Spectrum zpos, zneg, sigmap_t, sigma_tr, alphap;
float A;
```

The method to compute the reflectance takes the squared distance in the $z = 0$ plane between the point of incident illumination and the outgoing point of interest.

⟨*DiffusionReflectance Public Methods*⟩ +≡ 907
```
Spectrum operator()(float d2) const {
    Spectrum dpos = Sqrt(Spectrum(d2) + zpos * zpos);
    Spectrum dneg = Sqrt(Spectrum(d2) + zneg * zneg);
    Spectrum Rd = (1.f / (4.f * M_PI)) *
        ((zpos * (dpos * sigma_tr + Spectrum(1.f)) *
          Exp(-sigma_tr * dpos)) / (dpos * dpos * dpos) -
         (zneg * (dneg * sigma_tr + Spectrum(1.f)) *
          Exp(-sigma_tr * dneg)) / (dneg * dneg * dneg));
    return Rd.Clamp();
}
```

16.5.4 RENDERING WITH HIERARCHICAL INTEGRATION

The `DipoleSubsurfaceIntegrator::Li()` method performs standard direct lighting calculations for objects that don't have BSSRDFs. For those that do, the following fragment gets the medium's scattering properties from the `BSSRDF` object. The σ_a variable corresponds to the absorption coefficient introduced in Section 11.1, and σ_s' and σ_t' are the reduced scattering coefficient and reduced extinction coefficient, respectively, from Secton 16.5.3.

⟨*Evaluate BSSRDF and possibly compute subsurface scattering*⟩ ≡
```
BSSRDF *bssrdf = isect.GetBSSRDF(ray, arena);
if (bssrdf && octree) {
    Spectrum sigma_a  = bssrdf->sigma_a();
    Spectrum sigmap_s = bssrdf->sigma_prime_s();
    Spectrum sigmap_t = sigmap_s + sigma_a;
    if (!sigmap_t.IsBlack()) {
        ⟨Use hierarchical integration to evaluate reflection from dipole model 910⟩
    }
}
```

The R_d function defined in Equation (16.6) computes the diffuse subsurface reflectance between two points on the boundary of a semi-infinite scattering medium. We will now show how this function can be used to efficiently evaluate the subsurface scattering equation with a hierarchical integration scheme.

We start with a BSSRDF that is based on the diffusion subsurface reflectance approximation R_d,

$$S(p_o, \omega_o, p_i, \omega_i) = \frac{1}{\pi} F_t(\eta_o, \omega_o) R_d(\|p_i - p_o\|) F_t(\eta_i, \omega_i).$$

The division by π converts diffuse reflectance to a BSSRDF (though one that is a uniform function of direction), similar to how a BRDF's reflectance can be converted into a Lambertian BRDF (Section 8.1.1). The two Fresnel terms in turn account for the transmission of light through the boundary of the medium at the entrance and exit.

BSSRDF 598
BSSRDF::sigma_a() 599
BSSRDF::sigma_prime_s() 599
Intersection::GetBSSRDF() 484
M_PI 1002
Spectrum 263
Spectrum::Exp() 265
Spectrum::IsBlack() 265
Spectrum::Sqrt() 265

If this approximation is substituted into the subsurface scattering equation (Equation (5.9)), we have

$$L_o(p_o, \omega_o) = \int_A \int_{\mathcal{H}^2(\mathbf{n})} \left(\frac{1}{\pi} F_t(\eta_o, \omega_o) R_d(\|p_i - p_o\|) F_t(\eta_i, \omega_i) \right) L_i(p_i, \omega_i) |\cos \theta_i| \, d\omega_i \, dA.$$

This can be immediately simplified a bit, with the constant and $F_t(\eta_o, \omega_o)$ terms moved out of the integral entirely, since they are neither functions of position nor incident direction ω_i. In a similar manner, we can move the R_d term into just the area integral, since it doesn't depend on incident direction:

$$L_o(p_o, \omega_o) = \frac{1}{\pi} F_t(\eta_o, \omega_o) \int_A R_d(\|p_i - p_o\|) \left[\int_{\mathcal{H}^2(\mathbf{n})} F_t(\eta_i, \omega_i) L_i(p_i, \omega_i) |\cos \theta_i| \, d\omega_i \right] dA.$$

Note that, if not for the Fresnel transmittance term $F_t(\eta_i, \omega_i)$, the integral over incident directions ω_i in brackets would be the irradiance at the point p_i (Equation (5.3)). Therefore, the next approximation is to approximate this integral by the product of the irradiance at the point and the diffuse Fresnel transmittance $F_{dt}(\eta_i)$,

$$\int_{\mathcal{H}^2(\mathbf{n})} F_t(\eta_i, \omega_i) \, L_i(p_i, \omega_i) \, |\cos \theta_i| \, d\omega_i \approx F_{dt}(\eta_i) \, E(p_i, \mathbf{n}).$$

For a homogeneous medium, as we're assuming here, η is a constant, so $\eta = \eta_o = \eta_i$. We can apply the irradiance approximation above and move the F_{dt} term out of the integral over area, giving

$$L_o(p_o, \omega_o) \approx \frac{1}{\pi} F_t(\eta, \omega_o) \, F_{dt}(\eta) \int_A R_d(\|p_i - p_o\|) \, E(p_i, \mathbf{n}) \, dA.$$

The remaining terms in the integral over area give the radiant exitance M_o (introduced in Section 5.4) at the point p_o due to subsurface scattering from irradiance at the surface.

$$M_o(p_o) = \int_A R_d(\|p_i - p_o\|) \, E(p_i, \mathbf{n}) \, dA$$

We can approximate this integral as a sum over j area elements, each centered at a point p_j with area A_j and average irradiance E_j, giving

$$M_o(p_o) \approx \sum_j R_d(\|p_j - p_o\|) \, E_j \, A_j. \quad [16.7]$$

Intuitively, the error from this approximation depends on the variation of the respective terms; for example, if the R_d term actually varies significantly over the area of an element, then only evaluating it at the single point p_j may introduce unacceptable error. See Exercises 16.15 and 16.16 for further discussion of this issue and two ways of addressing it.

With this approach to computing M_o, we have the expression for computing outgoing radiance due to subsurface scattering that is used in the implementation below.

$$L_o(p_o, \omega_o) \approx \frac{1}{\pi} F_t(\omega_o)\, F_{dt}(p_o)\, M_o(p_o). \quad [16.8]$$

The terms other than $M_o(p_o)$ are computed trivially, and $M_o(p_o)$ is computed using the approximation of Equation (16.7), which is evaluated with the `SubsurfaceOctreeNode::Mo()` method.

⟨*Use hierarchical integration to evaluate reflection from dipole model*⟩ ≡ 908

```
DiffusionReflectance Rd(sigma_a, sigmap_s, bssrdf->eta());
Spectrum Mo = octree->Mo(octreeBounds, p, Rd, maxError);
FresnelDielectric fresnel(1.f, bssrdf->eta());
Spectrum Ft = Spectrum(1.f) - fresnel.Evaluate(AbsDot(wo, n));
float Fdt = 1.f - Fdr(bssrdf->eta());
L += (INV_PI * Ft) * (Fdt * Mo);
```

The `Mo()` method is called first with the root node of the tree. This method returns the radiant exitance at the point `pt` due to all of the `IrradiancePoints` in the nodes beneath it. At each node, it uses the aggregate values for the points beneath it, computed earlier in `InitHierarchy()`, if the error from using the clustered values would be low. Otherwise, it either sums the M_o values returned by its children nodes, or, if at a leaf, sums the contributions from each of the `IrradiancePoints` the leaf stores.

Note that there is an inaccuracy lurking in this approach: two separate objects that exhibit subsurface scattering end up using irradiance samples from the surface of the other object. In some cases, this is reasonable—consider a marble chess board made out of two types of marble in a checkered pattern where each check is modeled as an individual shape. The irradiance arriving at one check is reasonable to use for computing radiant exitance from its neighbor. On the other hand, the case illustrated in Figure 16.17, where the two objects are separated by a wall, is one where it's completely incorrect to use the other object's irradiance values and a per-primitive octree would give a better result. In general, the fall-off of the R_d function generally helps mitigate the visual impact of this issue.

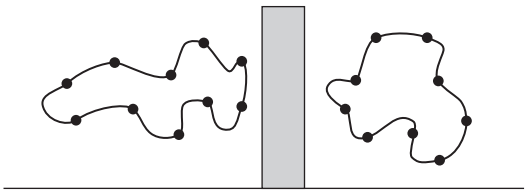


Figure 16.17: Setting That the Octree of Irradiance Points Doesn't Handle Well. With a single octree that holds all of the irradiance points in the scene, when we are computing radiant exitance from a point, we may inadvertently use irradiance samples from different objects. Here, the two objects are not only disconnected but also separated by a wall. The irradiance values on one's surface shouldn't affect the radiant exitance from the other, but the current implementation doesn't handle this case correctly.

BSSRDF::eta() 599
DiffusionReflectance 907
DipoleSubsurfaceIntegrator::octree 898
Fdr() 905
FresnelDielectric 437
FresnelDielectric::Evaluate() 437
INV_PI 1002
IrradiancePoint 898
Spectrum 263
SubsurfaceOctreeNode::InitHierarchy() 901
SubsurfaceOctreeNode::Mo() 911

⟨*DipoleSubsurfaceIntegrator Method Definitions*⟩ +≡
```
Spectrum SubsurfaceOctreeNode::Mo(const BBox &nodeBound, const Point &pt,
        const DiffusionReflectance &Rd, float maxError) {
    ⟨Compute Mo at node if error is low enough 911⟩
    ⟨Otherwise compute Mo from points in leaf or recursively visit children 911⟩
}
```

Two criteria are used to determine whether the clustered values at the node can be used. First, if the point being shaded is inside the node, then the clustered values are never used but the children are always visited (or the `IrradiancePoints` themselves are used, if in a leaf). The second criterion computes a conservative bound on the solid angle subtended by the area element as seen from the lookup point. If this value is less than a user-defined error bound and the lookup point is outside the node, then the clustered values at the node are used and recursion for the subtree beneath the node is skipped.

⟨*Compute M_o at node if error is low enough*⟩ ≡ 911
```
float dw = sumArea / DistanceSquared(pt, p);
if (dw < maxError && !nodeBound.Inside(pt))
    return Rd(DistanceSquared(pt, p)) * E * sumArea;
```

If the current node couldn't be used directly, then a finer approximation is used instead.

⟨*Otherwise compute M_o from points in leaf or recursively visit children*⟩ ≡ 911
```
Spectrum Mo = 0.f;
if (isLeaf) {
    ⟨Accumulate Mo from leaf node 911⟩
}
else {
    ⟨Recursively visit children nodes to compute Mo 911⟩
}
return Mo;
```

At leaf nodes, the best that can be done is to loop over the `IrradiancePoints` and apply Equation (16.7).

⟨*Accumulate M_o from leaf node*⟩ ≡ 911
```
for (int i = 0; i < 8; ++i) {
    if (!ips[i]) break;
    Mo += Rd(DistanceSquared(pt, ips[i]->p)) * ips[i]->E * ips[i]->area;
}
```

For interior nodes, the contributions of the non-`NULL` children are summed similarly.

⟨*Recursively visit children nodes to compute M_o*⟩ ≡ 911
```
Point pMid = .5f * nodeBound.pMin + .5f * nodeBound.pMax;
for (int child = 0; child < 8; ++child) {
    if (!children[child]) continue;
    BBox childBound = octreeChildBound(child, nodeBound, pMid);
    Mo += children[child]->Mo(childBound, pt, Rd, maxError);
}
```

BBox 70
BBox::Inside() 72
BBox::pMax 71
BBox::pMin 71
DiffusionReflectance 907
DistanceSquared() 65
IrradiancePoint 898
IrradiancePoint::area 898
IrradiancePoint::E 898
IrradiancePoint::p 898
octreeChildBound() 1026
Point 63
Spectrum 263
SubsurfaceOctreeNode::children 899
SubsurfaceOctreeNode::E 899
SubsurfaceOctreeNode::ips 899
SubsurfaceOctreeNode::isLeaf 899
SubsurfaceOctreeNode::Mo() 911
SubsurfaceOctreeNode::p 899
SubsurfaceOctreeNode::sumArea 899

16.5.5 SETTING SCATTERING PROPERTIES

It is remarkably unintuitive to set values of the absorption coefficient σ_a and the modified scattering coefficient σ_s' to achieve a desired visual result. If measured values of these parameters aren't available (for example, from the values made available from the `GetVolumeScatteringProperties()` utility function of Section 11.6.1), then the task for an artist trying to render subsurface scattering can be difficult.

Jensen and Buhler (2002) developed a different approach that derives scattering properties from a diffuse reflection color, the average distance light travels in the medium before scattering (the mean path length), and the index of refraction—much more intuitive parameters. The medium can be made more transparent by increasing the mean path length or more dense by decreasing it. The amount of multiple scattering can be controlled by how close the diffuse reflection color is to one.

The BRDF reflectance corresponding to a BSSRDF can be found by computing the integral of R_d to infinity; this gives the reflectance that results from a uniformly illuminated infinite surface with subsurface scattering. There is a compact closed form expression for this integral,

$$\int_0^\infty R_d(x)\,dx = \frac{\alpha'}{2}\left(1 + e^{-\frac{4}{3}A\sqrt{3(1-\alpha')}}\right) e^{-\sqrt{3(1-\alpha')}}, \qquad (16.9)$$

with $\alpha' = \sigma_s'/\sigma_t'$ and A defined as in Section 16.5.3.

The `RdIntegral()` function returns the value of the integral for given α' and A values.

⟨*Volume Scattering Local Definitions*⟩ ≡
```
static float RdIntegral(float alphap, float A) {
    float sqrtTerm = sqrtf(3.f * (1.f - alphap));
    return alphap / 2.f * (1.f + expf(-4.f/3.f * A * sqrtTerm)) *
        expf(-sqrtTerm);
}
```

If we consider for now a single wavelength of light with given reflectance and A value, we'd like to find the α' value such that the integral of Equation (16.9) returns the same reflectance. The value of the integral is monotonic and smooth so that a binary search quickly converges to an accurate result.

⟨*Volume Scattering Local Definitions*⟩ +≡
```
static float RdToAlphap(float reflectance, float A) {
    float alphaLow = 0., alphaHigh = 1.f;
    float kd0 = RdIntegral(alphaLow, A);
    float kd1 = RdIntegral(alphaHigh, A);
    for (int i = 0; i < 16; ++i) {
        float alphaMid = (alphaLow + alphaHigh) * 0.5f;
        float kd = RdIntegral(alphaMid, A);
        if (kd < reflectance) { alphaLow = alphaMid;  kd0 = kd; }
        else                  { alphaHigh = alphaMid; kd1 = kd; }
    }
    return (alphaLow + alphaHigh) * 0.5f;
}
```

GetVolumeScattering Properties() 600
RdIntegral() 912

We can put this all together into a small utility function that takes given diffuse reflectance, mean path length, and index of refraction values and returns σ_a and σ_s'. It works one component at a time, after converting to RGB.

⟨*Volume Scattering Definitions*⟩ +≡
```
void SubsurfaceFromDiffuse(const Spectrum &Kd, float meanPathLength,
        float eta, Spectrum *sigma_a, Spectrum *sigma_prime_s) {
    float A = (1.f + Fdr(eta)) / (1.f - Fdr(eta));
    float rgb[3];
    Kd.ToRGB(rgb);
    float sigma_prime_s_rgb[3], sigma_a_rgb[3];
    for (int i = 0; i < 3; ++i) {
        ⟨Compute α′ for RGB component, compute scattering properties 913⟩
    }
    *sigma_a = Spectrum::FromRGB(sigma_a_rgb);
    *sigma_prime_s = Spectrum::FromRGB(sigma_prime_s_rgb);
}
```

Given the modified albedo α' that corresponds to the reflectance value given, the rest of the scattering properties can be computed. The inverse mean path length is approximately the effective transport coefficient σ_{tr} and we can use the relationship between σ_{tr}, σ_t', and α' from Section 16.5.3 to solve for σ_t'. Finally, the relationships $\sigma_s' = \alpha'\sigma_t'$ and $\sigma_a = \sigma_t' - \sigma_s'$ are used to compute σ_s' and σ_a.

⟨*Compute α′ for RGB component, compute scattering properties*⟩ ≡ 913
```
float alphap = RdToAlphap(rgb[i], A);
float sigma_tr = 1.f / meanPathLength;
float sigma_prime_t = sigma_tr / sqrtf(3.f * 1.f - alphap);
sigma_prime_s_rgb[i] = alphap * sigma_prime_t;
sigma_a_rgb[i] = sigma_prime_t - sigma_prime_s_rgb[i];
```

FURTHER READING

Lommel (1889) was apparently the first to derive the equation of transfer, in a not widely known paper. Not only did he derive the equation of transfer, but he also solved it in some simplified cases in order to estimate reflection functions from real-world surfaces (including marble and paper) and compared his solutions to measured reflectance data from these surfaces.

Seemingly unaware of Lommel's work, Schuster (1905) was the next researcher in radiative transfer to consider the effect of multiple scattering. He used the term *self-illumination* to describe the fact that each part of the medium is illuminated by every other part of the medium, and he derived differential equations that described reflection from a slab along the normal direction assuming the presence of isotropic scattering. The conceptual framework that he developed remains essentially unchanged in the field of radiative transfer.

Soon thereafter, Schwarzschild (1906) introduced the concept of radiative equilibrium, and Jackson (1910) expressed Schuster's equation in integral form, also noting that

Fdr() 905
RdToAlphap() 912
Spectrum 263
Spectrum::FromRGB() 277
Spectrum::ToRGB() 275

"the obvious physical mode of solution is Liouville's method of successive substitutions" (i.e., a Neumann series solution). Finally, King (1913) completed the rediscovery of the equation of transfer by expressing it in the general integral form. Yanovitskij (1997) traced the origin of the integral equation of transfer to Chvolson (1890), but we have been unable to find a copy of this paper.

Blinn (1982b) first used basic volume scattering algorithms for computer graphics. The equation of transfer was first introduced to graphics by Kajiya and Von Herzen (1984). Rushmeier (1988) was the first to compute solutions of it in a general setting. Arvo (1993) first made the essential connections between previous formalizations of light transport in graphics and the equation of transfer and radiative transfer in general.

Other early work in volume scattering for computer graphics includes work by Max (1986) and Nishita, Miyawaki, and Nakamae (1987). Glassner (1995) provided a thorough overview of this topic and previous applications of it in graphics, and Max's survey article (Max 1995) also covers the topic well. One key application of volume scattering algorithms in computer graphics has been simulating atmospheric scattering. Work in this area includes papers by Klassen (1987) and Preetham, Shirley, and Smits (1999), who introduced a physically rigorous and computationally efficient atmospheric and skylighting model.

Researchers have recently had success in deriving closed-form expressions that describe scattering in participating media under certain assumptions; these approaches can be substantially more efficient than previous methods. See Sun et al. (2005), Pegoraro and Parker (2009), and Pegoraro et al. (2009) for examples of such methods. (Remarkably, Pegoraro and collaborators' work provides a closed-form expression for scattering from a point light source along a ray passing through homogeneous participating media with anisotropic phase functions.)

Rushmeier and Torrance (1987) used finite-element methods for rendering participating media, and Lafortune and Willems (1996) applied bidirectional path tracing to the problem. Other work includes Bhate and Tokuta's approach based on spherical harmonics (Bhate and Tokuta 1992) and Blasi et al.'s two-pass Monte Carlo algorithm, where the first pass shoots energy from the lights and stores it in a grid and the second pass does final rendering using the grid to estimate illumination at points in the scene (Blasi, Saëc, and Schlick 1993). More recently, Szirmay-Kalos et al. (2005) precomputed interacitions between sample points in the medium in order to more quickly compute multiple scattering. See Cerezo et al. (2005) for an extensive survey of approaches to rendering participating media.

Jensen and Christensen (1998) generalized the photon mapping algorithm for rendering participating media, and Jarosz et al. (2008) showed how to express the scattering integral over a beam through the medium as a measurement to be evalauted with particles from a particle-tracing algorithm (a similar measurement-based approach was introduced for reflection from surfaces in Section 15.6.1). This led to a much more efficient implementation of volume scattering. Pauly, Kollig, and Keller (2000) generalized the Metropolis light transport algorithm to include volume scattering. Pauly's thesis (Pauly 1999) described the theory and implementation of bidirectional and Metropolis-based algorithms for volume light transport. See Pegoraro et al. (2008b) for an interesting approach for

improving Monte Carlo rendering of participating media by using information from previous samples to guide future sampling.

Moon et al. (2007) made the important observation that some of the assumptions underlying the use of the equation of transfer—that the scattering particles in the medium aren't too close together so that scattering events can be considered to be statistically independent—aren't in fact true for interesting scenes that include small crystals, ice, or piles of many small glass objects. They developed a new light transport algorithm for these types of *discrete random media* based on composing precomputed scattering solutions.

There are a number of applications of visualizing volumetric data sets for medical and engineering applications. This area is called *volume rendering*. In many of these applications, radiometric accuracy is substantially less important than developing techniques that help make structure in the data apparent (e.g., where the bones are in CT scan data). Early papers in this area include those by Levoy (1988, 1990a, 1990b) and Drebin, Carpenter, and Hanrahan (1988).

Subsurface scattering was first introduced to graphics by Hanrahan and Krueger (1993), although their approach did not attempt to simulate light that entered the object at points other than at the point being shaded. Dorsey et al. (1999) applied photon maps to simulating subsurface scattering that did include this effect, and Pharr and Hanrahan (2000) introduced an approach based on computing BSSRDFs for arbitrary scattering media by integrating over the medium.

Kajiya and Von Herzen (1984) first introduced the diffusion approximation to graphics, though Stam (1995) was the first to clearly identify many of its advantages for rendering. See Ishimaru's book (1978) or Donner's thesis (2006) for the derivation of the diffusion approximation and Wyman et al. (1989) for the introduction of the principle of similarity.

The dipole approximation for subsurface scattering was developed by Farrell et al. (1992). It was introduced to computer graphics by Jensen et al. (2001b). Jensen and Buhler (2002) developed the hierarchical integration approach implemented here in the `DipoleSubsurfaceIntegrator`. The dipole approximation has since been the basis of a number of fast implementations for scan-line and interactive rendering (Hery 2003; Hao, Baby, and Varshney 2003; Dachsbacher and Stamminger 2003). Arbree et al. (2008) observed that the two-pass hierarchical integration approach doesn't scale well to complex scenes: excessive time may be spent generating irradiance samples in parts of the scene that aren't needed for rendering the final image. They proposed a one-pass approach based on the lightcuts method that generates irradiance samples on demand, clustering them for efficient integration. (See the "Further Reading" section in Chapter 15 for further information on lightcuts.)

Contini et al. (1997) generalized the dipole approach to *multipoles* to more accurately model finite scattering slabs. This approach was applied to subsurface scattering by Donner and Jensen (2005). However, even the multipole approach doesn't handle all types of scattering media well; the asssumptions of heterogeneous media and relatively high albedos are too restrictive for many interesting objects. Li et al. (2005) developed a hybrid approach that handles the first few bounces of light with Monte Carlo path tracing

DipoleSubsurfaceIntegrator 887

but then switches to a dipole approximation. Tong et al. (2005), Haber et al. (2005), and Wang et al. (2008b) further generalized the media supported, solving the diffusion equation on a grid of sample points. Fattal (2009) applied the discrete ordinates method, addressing a number of shortcomings of the direct application of that technique.

Donner et al. (2009) computed BSSRDFs with Monte Carlo simulation for a variety of scattering properties (phase function, scattering coefficients, etc.), and fit the resulting data to a low-dimensional model. This model much better accounts for the directional variation of scattered light and the properties of medium-albedo media.

Rendering realistic human skin is a challenging problem; this problem has driven the development of a number of new methods for rendering subsurface scattering after the initial dipole work as issues of modeling the layers of skin and computing more accurate simulations of scattering between layers have been addressed. For a good overview of these issues, see Igarashi et al.'s (2007) survey on the scattering mechanisms inside skin and approaches for measuring and rendering skin. Notable research in this area includes papers by Donner and Jensen (2006), d'Eon et al. (2007), Ghosh et al. (2008), and Donner et al. (2008). Donner's thesis includes a discussion of the importance of accurate spectral representations for high-quality skin rendering (Donner 2006, Section 8.5).

The Poisson sphere point distribution technique in the `DipoleSubsurfaceIntegrator` is based on the approach described by Lehtinen et al. (2008). Arbree et al. (2008) used a similar approach to distribute points for subsurface scattering. Bikker and Reijerse (2009) developed a related technique that tries to adapt the point distribution, placing more samples in areas where the illumination is changing rapidly. A much more efficient approach for placing Poisson disk-distributed points on surfaces was described by Cline et al. (2009a), though they require that a number of operations can be performed for each type of shape used. Their paper also has a good survey of previous work in this area.

EXERCISES

❷ 16.1 With inhomogeneous volume regions, where the optical depth between two points must be computed with ray marching, the `SingleScatteringIntegrator` volume integrator may spend a lot of time finding the attenuation between lights and points on rays where single scattering is being computed. One approach to reducing this computation is to take advantage of the fact that the amount of attenuation for nearby rays is generally smoothly varying and to use a precomputed approximation to the attenuation. For example, Kajiya and Von Herzen (1984) computed the attenuation to a directional light source at a grid of points in 3D space and then found attenuation at any particular point by interpolating among nearby grid samples. A more memory-efficient approach was developed by Lokovic and Veach (2000) in the form of deep shadow maps, based on a clever compression technique that takes advantage of the smoothness of the attenuation. Implement one of these approaches in `pbrt` and measure how much it speeds up the `SingleScatteringIntegrator` integrator. Under what sort of situations do approaches like these result in image errors?

DipoleSubsurfaceIntegrator 887
SingleScatteringIntegrator 882

● 16.2 Another effective method to speed up the SingleScatteringIntegrator is to eliminate the need to trace shadow rays from points along the ray passed to the SingleScatteringIntegrator::Li() method by computing shadow maps from each light source (Williams 1978; Reeves, Salesin, and Cook 1987) and use them to determine light source visibility in place of shadow rays. Modify pbrt to optionally use this approach, and measure the performance difference. What resolution do you find to be necessary in the shadow maps for high-quality results?

● 16.3 One shortcoming of the VolumeIntegrators in this chapter is that they always take a fixed step size along the ray through the participating medium. If the medium is very dense in some parts but sparse in others, this may be inefficient, as a short step size is needed to accurately resolve detail in the thick parts but is wasteful in the rest of the volume. Another way that this integration can be performed is to sample a beam transmittance at which to compute a scattering event, find the point along the ray where it has passed through that thickness, and then do the scattering computation there. For constant-density volumes, the corresponding offset along the ray can be found in closed form. For varying density volumes, ray marching will still in general be required, though with less overhead than the current implementation, since a full lighting calculation isn't being performed at each ray-marching point.

Add a method to the VolumeRegion class that supports sampling a given optical thickness along a ray and implement it for the various implementations. Derive the closed-form method to sample the distance at which a given beam transmittance value is reached for a homogeneous medium, and implement a ray-marching algorithm that computes this for general volumes. Then, modify the SingleScatteringIntegrator to use it. What is the performance difference when using your new approach, for rendering images of approximately equal quality as the original implementation?

● 16.4 Performing ray-marching to sample the point at which a desired beam transmittance value is reached as in Exercise 16.3 can still be computationally expensive. Coleman (1968) shows that if the maximum scattering coefficient of the medium is known, then an unbiased approach is to generate a candidate sample as if the medium was homogeneous, with a scattering coefficient equal to the maximum value, and then to probabilistically accept the candidate point and compute scattering with probability equal to the ratio of the actual scattering coefficient at the point and the maximum scattering coefficient. (This approach was introduced to graphics by Matthias Rabb (Shirley 2009).) Implement this technique and evaluate its effectiveness.

● 16.5 An additional advantage of the approaches in Exercises 16.3 and 16.4 compared to the emission-only and single scattering integrator implementations in this chapter is that the points at which scattering is computed end up being exponentially distributed—they are more likely to be close to the start of the ray, where less attenuatation has occurred and thus the contributions of the samples taken will have been reduced less by the beam transmittance term. However,

SingleScatteringIntegrator 882
SingleScatteringIntegrator::Li() 882
VolumeIntegrator 876
VolumeRegion 587

there are cases where this isn't the optimal approach, either. Consider a medium where there is no illumination in the front, as seen by the camera, but an extremely bright light is shining in the back part of it. Even though the points in the back will be attenuated more, they're the only ones where the scattering term will be nonzero!

The general problem is that we need to compute a Monte Carlo estimate of a function that's comprised of the product of two functions: the transmittance and the source term. Each of them should be able to direct where sampling happens, with the goal that the combined function is sampled at points where the product of the two functions is large. The general problem here is similar to the case of sampling direct illumination at a surface from a small light source: sampling the BSDF is an inefficient approach, but sampling using the light works quite well. In some cases, sampling the ray is inefficient, but sampling using the light is effective. (As another example, consider a spotlight with a small cone angle shining light through a participating medium. The `SingleScatteringIntegrator` integrator will find it very difficult to efficiently render an image that accurately captures the beam, since a small step size will be needed to find points along rays that lie in the spotlight's cone, yet a small step size will be inefficient for most of the rays that don't pass through the cone at all (Nishita, Miyawaki, and Nakamae 1987).)

Modify `pbrt`'s light sampling interfaces and the `SingleScatteringIntegrator` implementation so that light sources are able to convey information about which parts of particular rays they potentially illuminate. One possibility would be to allow light sources to return a parametric t range to sample for each ray; a default implementation could set the range to $[0, \infty)$. Another possible approach would be to allow the lights the opportunity to sample points along rays directly. Implement one of these techniques in `pbrt` and measure the improvement in running time for difficult-to-render scenes like the one described earlier.

2 **16.6** Extend the `SingleScatteringIntegrator` to use multiple importance sampling based on sampling points on the light source and sampling the phase function for the direct lighting computation. Under what circumstances will this method give substantially less variance than the current implementation?

3 **16.7** Design and implement a general Monte Carlo path-tracing approach to compute images with multiple scattering in participating media. (To combat the high running times of these algorithms, it may be worthwhile to first implement one of the techniques for reducing the cost of shadow rays described in earlier exercises.) For background information and general approaches, it may be helpful to read Lafortune and Willems's paper about bidirectional path tracing in participating media, even if you don't implement a bidirectional algorithm (Lafortune and Willems 1996). Furthermore, Pauly, Kollig, and Keller (2000) derived the generalization of the path integral form of the light transport equation for the volume scattering case; their insights may be useful as well. Render images that show the effect of multiple scattering in participating media.

SingleScatteringIntegrator 882

❸ 16.8 Photon mapping has also been shown to be an effective way to model the effect of multiple scattering in participating media. Read Jensen and Christensen's paper on this topic (Jensen and Christensen 1998) and implement this method in a new `VolumeIntegrator` in `pbrt`. You may find that the improved method for sampling and interpolating the photons described by Jarosz et al. (2008) substantially improves the efficiency of your implementation.

❸ 16.9 Ren et al. (2008) described an approach for efficiently rendering participating media where both radiance and its gradient are computed on an adaptive grid and then radiance values are interpolated with these two quantities. Read their paper and implement this approach as a `VolumeIntegrator` in `pbrt`. How does efficiency compare to the `SingleScatteringIntegrator`?

❷ 16.10 Recall from Section 16.5.3 that the diffusion approximation doesn't include the effect of direct transmission through the medium and light that is scattered once in the medium. Modify the `DipoleSubsurfaceIntegrator::Li()` method so that it accounts for these two currently missing terms. For direct transmission, you will want to compute the refracted ray that enters the object and find the intersection point where it leaves the object. The incident radiance along the ray that refracts back out of the object should then be attenuated by the transmittance of the ray inside the object. (It may be worthwhile to use Russian roulette to avoid computing incident radiance along the exiting ray for cases where the transmittance is very low.)

For single scattering, you will want to sample one or more points along the segment inside the object and compute single scattering along the lines of the `SingleScatteringIntegrator`. There are two subtleties with an implementation in this context. First, it's important to sample scattering from an exponential distribution along the segment: because the attenuation coefficient is relatively large in media with high albedos, single scattering at points that aren't close to the ray's entry-point to the medium makes a low contribution to the final result. Second, you will need to ignore intersections with the object's surface for shadow rays traced from the scattering point. (One way to do this is to trace initial shadow rays until they intersect the surface and then trace a second ray to do the actual shadow occlusion computation.)

After implementing these changes, compare images rendered with and without these two terms. How much of a difference do they make in practice? For what sorts of models is the difference most evident?

❸ 16.11 Implement a subsurface scattering integrator that uses Monte Carlo path tracing to compute multiple scattering. Verify the correctness of your implementation by comparing results when rendering a semi-infinite slab—the case with the lowest expected error for the diffusion dipole approach. Recall that in high-albedo media, paths of hundreds or thousands of bounces may be necessary to compute accurate results. Render images that demonstrate cases where the dipole approximation introduces excessive error but Monte Carlo computes a correct result. How much slower is the Monte Carlo approach for cases where the dipole is accurate?

DipoleSubsurfaceIntegrator::Li() 908
SingleScatteringIntegrator 882
VolumeIntegrator 876

16.12 As discussed in Section 16.5.1, the random ray-tracing approach for placing Poisson-disk points on surfaces can be computationally expensive. Read the paper by Cline et al. (2009a) on a more efficient approach for placing Poisson disk points on surfaces and implement their approach. Modify the `DipoleSubsurfaceIntegrator` to loop over all primitives that have a BSSRDF (you'll need to add a method to the `Primitive` interface that reports this information), and modify the `Shape` interface as needed to implement their approach. How much faster is this approach than the one in the current implementation? Does it have any disadvantages?

16.13 Generalize the irradiance computation in the `DipoleSubsurfaceIntegrator` to create and use one of the other `Integrators` in `pbrt` so that indirect illumination is included in the computed irradiance values. As part of this, you will probably want to break the irradiance computation into tasks so that they can run in parallel. Render images that show the improvement from including indirect lighting with a test scene.

16.14 Approximations of the dipole model such that the medium is semi-infinite and homogeneous can lead to substantial errors when these conditions are not true. Read Donner and Jensen's paper (2005) on using multiple dipoles (multipoles) to render thin translucent materials and on new techniques to handle media with layers with differing scattering properties. Implement their method in `pbrt` and render images showing the improved results when rendering thin translucent objects.

16.15 With the `DipoleSubsurfaceIntegrator`'s hierarchical integration scheme, it can be necessary to generate irradiance samples at a high density on the surface to achieve good results. (Figure 16.18 shows the characteristic splotchy artifacts that result from an insufficient number of samples.) Langlands and Mertens (2007) observed that these artifacts are the result of the rapid falloff of the subsurface reflectance term R_d; when shading some points, there will be a nearby irradiance sample that will match up with the peak of R_d and make a large contribution, but at others there won't be any samples sufficiently close by. (Recall the graph of the R_d function in Figure 16.16.)

To address this problem, Langlands and Mertens proposed breaking the subsurface scattering computation into two terms: one for nearby points and one for the rest. The average irradiance, E_n, of the nearby points up to some distance d is computed; a weighting function of distance $w(x)$ blends between the contributions of near and far irradiance samples, and Equation (16.8) is generalized to:

$$L_o(p_o, \omega_o) \approx 2\pi E_n \int_0^d w(x)\, R_d(x)\, dx + \frac{1}{\pi} F_t(\omega_o)\, F_{dt}(p_o) \sum_j (1 - w(\|p_j - p_o\|)) R_d(\|p_j - p_o\|)\, E_j\, A_j.$$

Thus, farther away away points are handled using the usual approach. They suggest using a hat function $w(x) = \max(0, 1 - d)$ for the weighting function and

DipoleSubsurfaceIntegrator 887
Integrator 740
Primitive 185
Shape 108

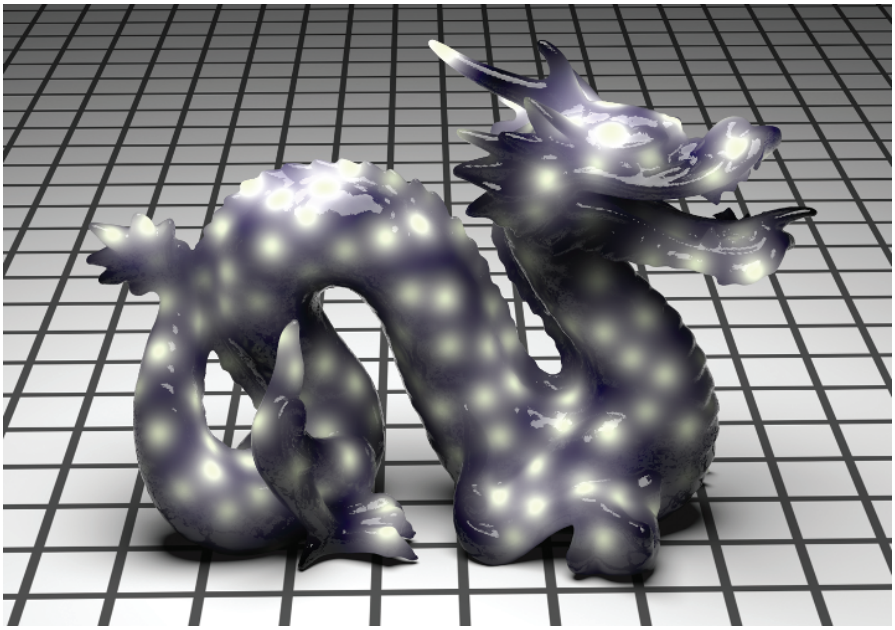


Figure 16.18: Characteristic Splotchy Artifacts from Sampling Irradiance at Too Low a Rate with the `DipoleSubsurfaceIntegrator`. The model from Figure 16.9 is rendered with not enough irradiance samples. The image has artifacts due to the rapid fall-off of the R_d function, such that there are no irradiance samples near its peak at many of the points being shaded. Exercises 16.15 and 16.16 outline solutions to this problem.

selecting the distance d so that it includes ten or so irradiance samples. For efficiency, the integral of the product of $w(x)$ and $R_d(x)$ should be precomputed and tabularized.

Implement this method in `pbrt`. How much lower an irradiance point sampling frequency can you use to get good results versus the current implementation?

❷ 16.16 Another approach for reducing the splotchy artifacts from the rapid R_d function fall-off was suggested by Neulander (2009). He introduced the idea of computing the incident irradiance at each outgoing point p_o where the hierarchical integration is being performed. Then, whichever point in the octree of irradiance points is closest to the outgoing point is replaced with a sample that has the same position and irradiance as the actual lookup point. This approach ensures that one sample is always taken at the peak of the subsurface reflectance function. Implement this approach for rendering subsurface scattering in `pbrt` and compare the results to the builtin dipole integrator.

❸ 16.17 Donner et al. (2009) performed extensive numerical simulation of subsurface scattering from media with a wide range of scattering properties and then computed coefficients to fit an analytical model to the resulting data. They have shown that rendering with this model is more efficient than full Monte Carlo integration, while handling well many cases where the approximations

DipoleSubsurfaceIntegrator 887

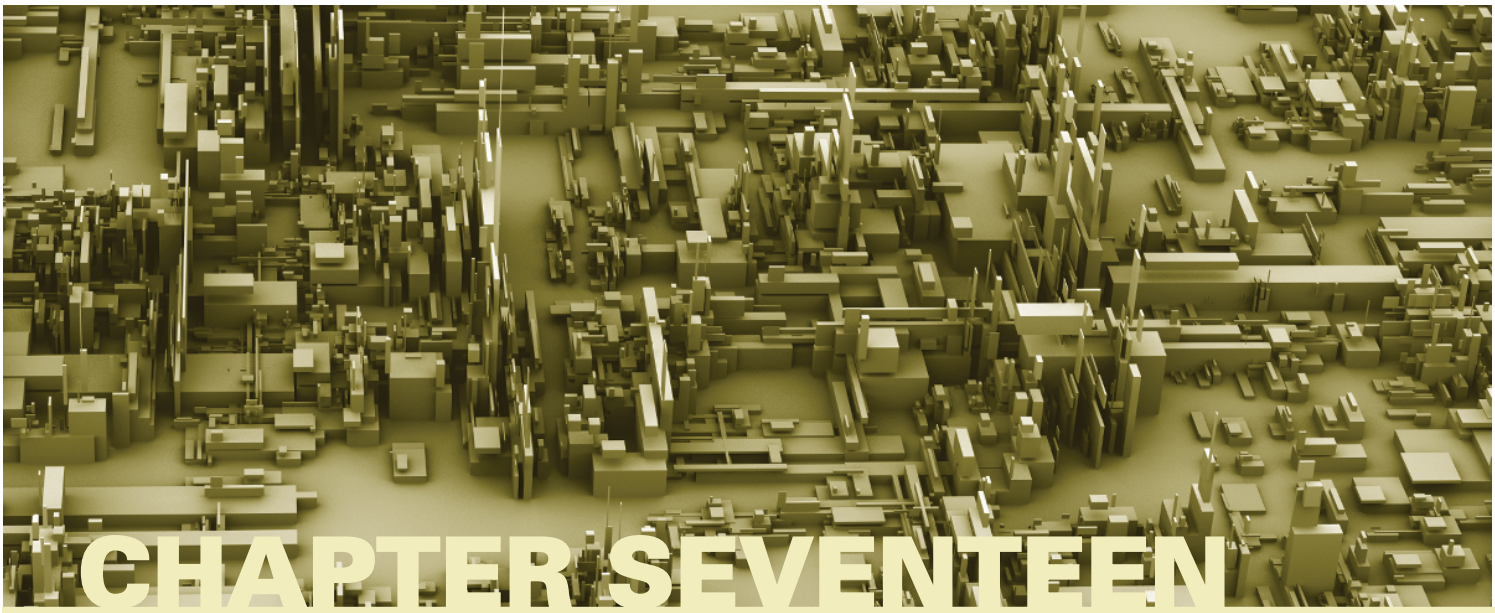
of the dipole model are unacceptable. For example, their model accounts for directional variation in the scattered radiance and handles media with low and medium albedos well. Read their paper and download the data files of coefficients. Implement a new subsurface integrator in `pbrt` that uses their model.

To implement this model, you will need a way to generate sample points on the surface around the point being shaded. One way to do this is to precompute a large number of points on the surface (along the lines of the `SurfacePointsRenderer`) and then store them in a spatial data structure like a kd-tree so that you can efficiently find points around a lookup point.

3 16.18 Modify `pbrt`'s interfaces and abstractions so that fully varying 3D scattering properties can be used for subsurface scattering. You may want to base your implementation on a grid of sample values, with the boundary defined by a `Primitive`'s surface. Then, implement a more general subsurface light transport simulation algorithm that handles 3D variation such as the one described by Tong et al. (2005), Haber et al. (2005), Wang et al. (2008b), or Fattal (2009).

SurfacePointsRenderer 888

This page intentionally left blank



CHAPTER SEVENTEEN

★ 17 LIGHT TRANSPORT III: PRECOMPUTED LIGHT TRANSPORT

The rendering algorithms in the preceeding chapters generally take minutes to hours of computation to generate high-quality imagery for interesting scenes. This is in general the price to be paid for their robustness, flexibility, and generality. Of course, this computational cost has a number of disadvantages. Artists modeling scenes generally desire quick feedback, and many applications require not just faster rendering but also full interactivity.

It's desirable that the images generated by interactive rendering share in the benefits of the improvements in light transport algorithms of recent years. A variety of *precomputed light transport* algorithms have been developed to this end: the general idea is to separate the complete rendering computation into a portion that can be performed offline in a preprocess and stored in a data structure, such that the results can then be incorporated into the final rendering computation later—for example, in an interactive system. As a simple example of this idea of reusing precomputed data in rendering, consider that the photon map integrator of Chapter 15 could be modified to save the photon map it generated to disk. When rendering a walkthrough of a scene (where only the camera position changed from frame to frame, and the objects, materials, and lights stayed the same), the same photon map could be reused across all of the frames of the animation, saving the cost of recomputing it.

An early example of a precomputation-based approach to rendering is a lighting design developed by Dorsey et al. (1995). Their goal was to build a system for opera lighting designers that would allow them to interactively change the colors and intensities of the light sources illuminating a set until a desired result was achieved. (The positions of the light sources themselves were fixed.) Because they were using finite-element light

transport algorithms to compute global illumination and because of the slow computers of the time, they had to develop a system that didn't rerender the scene from scratch each time a lighting parameter changed.

The system they built took advantage of the linearity of light transport; recall from Section 5.4 that the image that results from lighting a scene with two light sources will be the same as the sum of the two images from lighting them with each source independently. Therefore, they rendered one image for each light source, at unit intensity. The interactive system loaded all of the images into memory and then allowed the user to change the color and intensity of each light source, computing the resulting image with a weighted sum of the individual light source images.

Another early technique in precomputed light transport is *ambient occlusion* (AO) (Zhukov et al. 1998; Landis 2002). The basic idea is to precompute at points on surfaces how much of the hemisphere above each point is occluded by other objects in the scene and how much is not occluded, using the resulting value in the range $[0, 1]$ to modulate a standard ambient lighting term. To compute the ambient occlusion, at each point of interest, we estimate the value of the integral

$$a(\mathrm{p}, \mathbf{n}, d) = \frac{1}{\pi} \int_{\mathcal{H}^2(\mathbf{n})} V(\mathrm{p}, \omega_\mathrm{i}, d)\, |\cos \theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}, \qquad [17.1]$$

where $V(\mathrm{p}, \omega, d)$ is a visibility funcion that returns one if the ray from p in direction ω is not occluded up to the distance d, and zero if it is occluded. The resulting value can then be stored with the scene geometry and then be used later for efficient rendering: Figure 17.1 shows the idea for an abstract structure scene; the image shows a visualization of the ambient occlusion values computed at each point.

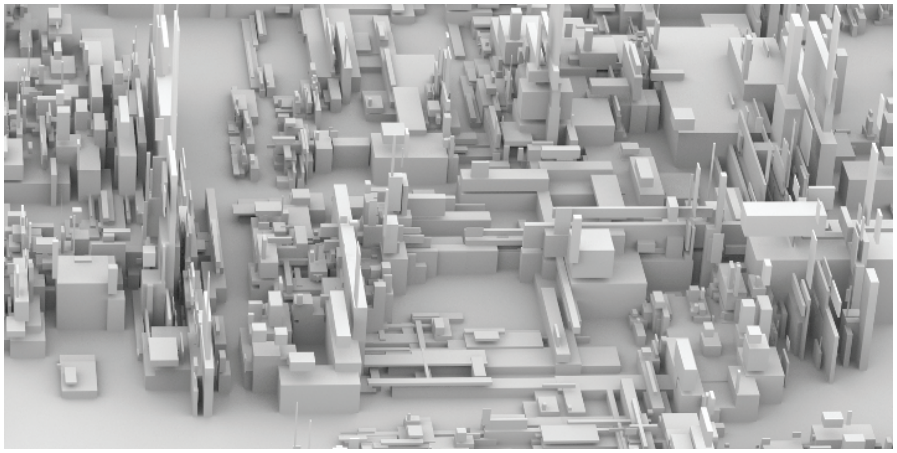


Figure 17.1: Ambient Occlusion. Abstract city-like scene rendered with ambient occlusion values, computed via Equation (17.1). Because the ambient occlusion value can be precomputed and the direct lighting can be computed very efficiently, a convincing facsimile of global illumination can be produced using AO values in shading calculations. (Model courtesy of Mark Schafer; this model was generated procedurally using *Structure Synth*.)

There are two main ways in which approaches to precomputed light transport can be categorized:

- What simplifications are made or which limitations are accepted, and hence which components of the simplified light transport model are precomputed?
- At what points in which domain are the precomputed values computed, and in what form are they stored?

For example, Dorsey et al.'s opera lighting algorithm limits the viewpoint to be a fixed position and requires that the scene geometry and light positions be fixed. These choices make it possible to precompute lighting values at the pixels of images and use them for rerendering.

A common use of ambient occlusion is for real-time rendering. In that setting, ambient occlusion values might be computed at points on surfaces of static geometry in the scene and stored in texture maps. When rendered in real time, the ambient occlusion values are retrieved from the textures and used for shading. Storing AO values in texture maps and using them for rendering in this manner does introduce the limitation that dynamic objects in the scene wouldn't affect the ambient occlusion of static objects and that dynamic objects won't have an ambient occlusion component of their own. More sophisticated approaches to ambient occlusion have been developed to address these sorts of shortcomings; see the "Further Reading" section for details.

Many different choices have been made with regard to the above two questions: for simplifications to the rendering problem, the viewpoint, geometry, materials, and light positions may or may not be fixed. Some of these may be constrained—for example, a limited set of viewpoints, a limited range of allowed motion of a light source, diffuse reflection only, etc. The precomputed values may in turn be stored in images, at polygon mesh vertices, texture maps, or 3D volumetric representations. Different requirements for quality and performance lead to different trade-offs in this space.

In the remainder of this chapter, we'll introduce some of the theory and a number of implementations of algorithms for precomputed light transport. This area of rendering has seen a substantial amount of research in recent years, so this chapter can only provide an introduction to some of the main ideas; the "Further Reading" section has many more references for this area. We will also side-step some of the important practical issues in representing the precomputed values (for example, parameterizing general triangle meshes for mapping (u, v) texture maps to them for storing precomputed values), instead focusing on how the precomputation is done by an offline renderer and showing how these values are then used for rendering without going into the details of their use in a particular real-time rendering environment.

17.1 BASIS FUNCTIONS: THEORY

To generalize a lighting design system, we might want to make it possible to move the position of light sources in the scene. For example, still assuming a fixed viewpoint, we'd ideally like to precompute a generalized image where each pixel stored a compact function of light position, such that evaluating the function for a given light position would give the color of the corresponding pixel for that light position.

The application of *basis functions* can make this sort of generalization possible. The idea here is to take a generally infinite-dimensional space of functions (pixel radiance as a function of light-source position) and represent them with a specific set of basis functions and corresponding basis function coefficients. Given a fixed preselected set of basis functions, the coefficient values at each pixel define the approximation to the general function in terms of the basis. The result will usually be an approximation, since a finite set of coefficients can't represent all possible functions. However, if the basis is chosen well, or the functions being approximated have properties like smoothness that can be taken advantage of, the results can be close enough for many applications.

Using basis functions in this manner makes it possible to represent potentially arbitrarily complex functions for rendering—for example, the light arriving at a point on a surface over the course of a day as the sun moves across the sky. There are three important advantages of representing functions like these with a basis that collectively make this technique extremely useful for interactive rendering.

The first advantage is that approximating a function with a basis makes it possible to map the infinite-dimensional space of functions to a finite-dimensional space of coefficients. While some functions can be represented precisely—for example, a constant function or the radiance arriving at a point from an unoccluded spherical light source—many others, like the directional distribution of incident radiance at a point in a complex environment, cannot. Representing complex functions like these with a basis transforms them to a form that is amenable to mathematical manipulation and evaluation in programs.

The second key advantage is the ability to compress the representation of the original function: if the error introduced from projection into the basis isn't objectionable, then basis functions can be used to substantially compress the representation of a function by converting it into a number of floating-point coefficients.

The third advantage is efficiency: certain operations may be much more efficiently performed with functions represented in a basis than with the original functions. In particular, *orthonormal* basis functions, which will be introduced in the following section, have important properties that make integration of products of functions in the basis very efficient. Another example of this idea is the use of the Fourier transform for image processing: operations like blurring an image with a very wide kernel may be much more efficient in the Fourier domain than in the original image domain.

There are many trade-offs to consider in choosing a set of basis functions for a particular application; how efficiently they can be evaluated, how easy it is to compute the representation of a given function in the basis, and how well the basis captures important features of the function, including discontinuities and smoothness.

17.1.1 A PIECEWISE-CONSTANT BASIS

A straightforward example of using basis functions to approximate an arbitrary function is to use a set of *piecewise-constant* basis functions, where we might decide to use four basis functions over the range [0, 1]. The first basis function, $B_1(x)$, has the value one between zero and 1/4 and is zero elsewhere; the next one, $B_2(x)$, is one between 1/4 and 2/4; and so forth.

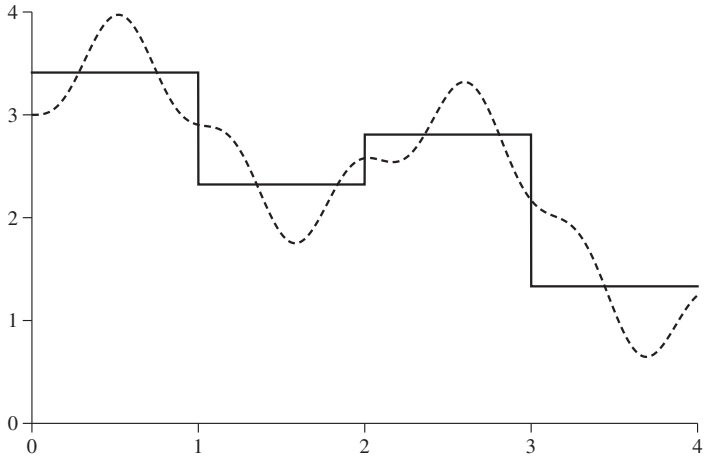


Figure 17.2: Representation of a Function in a Simple Piecewise-Constant Basis. A function (dashed lines) represented in the basis function defined by Equation (17.2) (solid lines).

$$B_i(x) = \begin{cases} 1 & \frac{i-1}{n} \leq x < \frac{i}{n} \\ 0 & \text{otherwise.} \end{cases} \tag{17.2}$$

To represent an arbitrary function $f(x)$ in this basis, we might evaluate it at a series of positions at the midpoint of each basis function's range, x_i, $x_0 = 1/8$, $x_1 = 3/8$, and so forth, giving a series of basis coefficient values $c_i = f(x_i)$. The representation of the original function in the basis, $\tilde{f}(x)$, is then found by a weighted sum of the basis functions (Figure 17.2):

$$f(x) \approx \tilde{f}(x) = \sum_i c_i B_i(x).$$

A more accurate approximation to f might be found by computing its average value over each subrange [0, 1/4), [1/4, 1/2), etc., or the approximation could be improved with a finer decomposition of [0, 1] with more basis functions. Alternatively, if we knew that f was a smooth function and that preserving the smoothness in the approximated function was important, we might use a different basis entirely (such as the Fourier basis).

In general, the representation of a function in a basis $\tilde{f}(x)$ and the original function $f(x)$ will be different. Given a particular basis, some functions can be represented exactly (for example, with the piecewise-constant basis used in the example above, the constant function $f(x) = c$). In general, however, the representation of the function in the basis will not match the original function exactly.

This notion of error in the approximation is made rigorous by defining a *norm* to measure the error between two functions in a space:

$$L_n[f(x), g(x)] = \left[\int (f(x) - g(x))^n \, dx\right]^{\frac{1}{n}}.$$

Different choices of n for the L_n norm measure different types of error; for example, L_1 gives the average error, L_2 reduces the penalty for small errors, and the L_∞ norm gives the maximum error over the domain, penalizing isolated large errors.

17.1.2 PROJECTION ONTO A BASIS

In the general case, we are given some function $f(x)$ and have selected a set of N basis functions $B_i(x)$ defined over a domain D. (Here we are using the notation $f(x)$ to represent more than just functions over the 1D reals, but instead functions over arbitrary-dimensional domains.) We then want to compute the coefficients c_i that represent the function in the basis. The approximated function is given by the weighted sum of basis functions

$$\tilde{f}(x) = \sum_i^N c_i B_i(x).$$

In particular, we would like a set of c_i such that they minimize the difference between the original function $f(x)$ and the approximated function $\tilde{f}(x)$.

It can be shown that the optimal coefficients in terms of minimizing the least-squares error between $f(x)$ and $\tilde{f}(x)$ are given by integrating $f(x)$ with the *dual* of the basis, $\tilde{B}_i(x)$:

$$c_i = \int_D f(x)\tilde{B}_i(x)\,\mathrm{d}x, \tag{17.3}$$

where the dual of a basis function is defined as the function $\tilde{B}_i(x)$ such that

$$\int_D B_i(x)\tilde{B}_j(x)\,\mathrm{d}x = \delta_{i,j},$$

where $\delta_{i,j}$ is the Kronecker delta function,

$$\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise.} \end{cases}$$

17.1.3 ORTHONORMAL BASIS FUNCTIONS

An important type of basis functions are *orthonormal basis functions*. An orthonormal basis has two important properties. The first is *orthogonality*: given any two distinct basis functions $B_i(x)$ and $B_j(x)$ where $i \neq j$, the integral of the product of the two basis functions is zero:

$$\int_D B_i(x)B_j(x)\,\mathrm{d}x = 0.$$

Intuitively, this property can be understood to mean distinct basis functions account for distinct properties of the function being represented in the basis. Basis functions like the piecewise-constant basis used in Section 17.1.1 naturally have this property, since the product $B_i(x)B_j(x)$ is always zero for $i \neq j$ and thus the integral of their product is zero. Note, however, that many orthonormal bases do not have this property of non-overlapping domains although their products still integrate to zero over the entire domain.

The second property of an orthonormal basis is a normalization property—the integral of a particular basis function with itself over the domain is one:

$$\int_D B_i(x) B_i(x)\, \mathrm{d}x = 1.$$

These properties can be understood as generalizations of how the x, y, and z coordinate axes form an orthonormal basis for points in 3D space: given any particular point in 3D, there's a unique set of coordinates that represents it.

Given these definitions, the piecewise-constant basis defined above is orthogonal but not orthonormal (unless $n = 1$)—the value of the integral of each one's product with itself over the domain is $1/n$. However, the piecewise-constant basis above can be easily modified to be normalized

$$B_i(x) = \begin{cases} n & \frac{i-1}{n} \leq x < \frac{i}{n} \\ 0 & \text{otherwise.} \end{cases}$$

Using an orthonormal basis to represent a space of functions has two important advantages compared to using a non-orthonormal basis: projecting a function into the basis is easier, and integration of the product of functions in the basis can be computed extremely efficiently.

The first advantage—easy projection of functions into the basis—comes from the useful feature of orthonormal basis functions that each basis function is equal to its dual:

$$B_i(x) = \tilde{B}_i(x).$$

The effect of this property is that coefficients of an arbitrary function in the basis are found by just integrating against the basis functions themselves:

$$c_i = \int_D f(x) B_i(x)\, \mathrm{d}x.$$

There's no need to find the dual basis functions for the chosen basis.

The second advantage, efficient integration, is an extremely useful one in practice. We will often want to compute the integral of the product of two functions represented in the basis (for example, a BRDF function and an incident radiance function). If the two functions $f(x)$ and $g(x)$ are represented by two sets of coefficients, c_i and c_i', in the basis, the integral is

$$\int_D \tilde{f}(x) \tilde{g}(x)\, \mathrm{d}x = \int_D \left(\sum_i c_i B_i(x) \right) \left(\sum_j c_j' B_j(x) \right) \mathrm{d}x.$$

The sums and the coefficients can be pulled outside of the integral, giving a nested sum of integrals of products of basis functions:

$$\sum_i \sum_j c_i\, c_j' \int_D B_i(x) B_j(x)\, \mathrm{d}x.$$

Recall that orthonormality means that for $i \neq j$, $\int B_i(x)B_j(x)\mathrm{d}x = 0$. Therefore, many terms in the above sum are necessarily zero, and what remains is:

$$\sum_i c_i \, c'_i \int_D B_i(x)B_i(x) \, \mathrm{d}x.$$

Finally, because the integral of the product of an orthonormal basis function with itself is one, the integrals are all one, and we are left with:

$$\int_D \tilde{f}(x)\tilde{g}(x) \, \mathrm{d}x = \sum_i c_i \, c'_i. \tag{17.4}$$

This is an incredibly useful result: the integral of the two functions in the basis can be found as a simple inner product of their coefficients. Unfortunately, integrals of products of three or more functions in orthonormal bases aren't as efficient as products of just two functions; the "Further Reading" section has pointers and information about these triple-product integrals, which have many important applications in rendering but are substantially more computationally complex.

17.2 SPHERICAL HARMONICS

The spherical harmonic (SH) basis functions are an orthonomal basis defined over the unit sphere; they are essentially a generalization of the Fourier basis to that domain (Section 7.1.1). Recall that the Fourier transform makes it possible to express many functions as weighted sums of sinusoids; the SH basis can do the same for directionally varying functions on the sphere and are thus a useful basis for many rendering problems. The SH basis will be used for all of the precomputed light transport algorithms in the remainder of this chapter.

The spherical harmonic functions $Y_l^m(\theta, \phi)$ are defined over a series of $l_{\max}$ *bands*, where each band l has indices $m = -l_{\max}, \ldots, l_{\max}$. Thus, if we wanted to use bands up to $l_{\max} = 1$ to represent a function in the SH basis, we would have basis functions $Y_0^0(\theta, \phi)$, $Y_1^{-1}(\theta, \phi)$, $Y_1^0(\theta, \phi)$, and $Y_1^1(\theta, \phi)$. The basis functions are functions of direction (θ, ϕ) defined using the associated Legendre polynomials $P_l^m(x)$.

$$Y_l^m(\theta, \phi) = K_l^m \mathrm{e}^{\mathrm{i}m\phi} P_l^{|m|}(\cos\theta). \tag{17.5}$$

Here, K_l^m is a normalization factor,

$$K_l^m = \sqrt{\frac{2l+1}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}, \tag{17.6}$$

and the first few associated Legendre polynomials are $P_0^0(x) = 1$, $P_1^{-1}(x) = 1/2\sqrt{1-x^2}$, $P_1^0(x) = x$, and $P_1^1(x) = -\sqrt{1-x^2}$. In the implementation below, we will compute values of the associated Legendre polynomials using a series of recurrences that express the higher order polynomials in terms of the lower order ones.

For rendering applications, the *real spherical harmonics* are typically used; the imaginary factor i in Equation (17.5) is not necessary. The real spherical harmonics are defined as:

$$Y_l^m(\theta, \phi) = \begin{cases} \sqrt{2} K_l^m P_l^{-m}(\cos\theta) \sin(-m\phi) & m < 0 \\ K_l^m P_l^m(\cos\theta) & m = 0 \\ \sqrt{2} K_l^m P_l^m(\cos\theta) \cos(m\phi) & m > 0. \end{cases} \quad [17.7]$$

These equations can also be expressed as polynomials in Cartesian (x, y, z) coordinates on the unit sphere rather than (θ, ϕ) angles. The polynomial forms can be found by applying the identities we previously used for BRDF direction vectors on page 425. For example,

$$Y_1^1(x, y, z) = -\sqrt{\frac{3}{4\pi}} x.$$

The basis projection formula from Equation (17.3) gives us the formula to compute the coefficients of the projection of a given function $f(\omega)$ into the SH basis (recall that the SH basis is orthonormal, so the basis functions are equal to their duals):

$$c_l^m = \int_{S^2} f(\omega) Y_l^m(\omega) \, d\omega. \quad [17.8]$$

Given SH coefficients, the reconstructed function from the coefficients is

$$\tilde{f}(\omega) = \sum_{l=0}^{l_{\max}} \sum_{l=-m}^{m} c_l^m Y_l^m(\omega). \quad [17.9]$$

It's often useful to treat a set of SH coefficients or SH basis functions as a vector indexed by a one-dimensional value i rather than indexed by (l, m):

$$c_0 = c_0^0, \qquad c_1 = c_1^{-1}, \qquad c_2 = c_1^0, \qquad c_i = \ldots \quad [17.10]$$

In general, the coefficient (l, m) maps to the index $i = l^2 + l + m$.

We will use Monte Carlo integration to project functions into the SH basis. Using the standard Monte Carlo estimator and the indexed notation, if N random samples are taken, we have the coefficients c_i given by

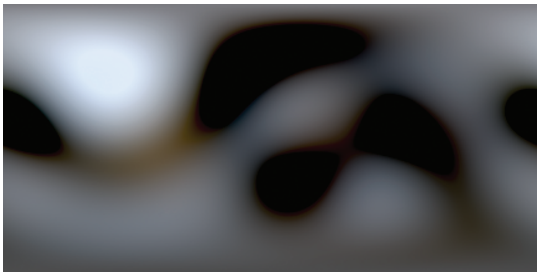
$$c_i \approx \frac{1}{N} \sum_{j=1}^{N} \frac{f(\omega_j) Y_i(\omega_j)}{p(\omega_j)}. \quad [17.11]$$

In implementations, a single set of samples ω_j will generally be generated for all c_i estimates, making it possible to reuse the computed values of $f(\omega_j)$ and $p(\omega_j)$ in all of their estimates, and only needing to evaluate $Y_i(\omega_j)$ for each of the coefficients c_i.

Figure 17.3 shows an example of representing an environment map in the SH basis for increasing numbers of basis function coefficients; note that as more coefficients are used, the approximation becomes closer to the original image. Further, note the smoothness of the function reconstructed from the SH basis. For many rendering tasks involving diffuse or very glossy reflection, very smooth representations of illumination such as these can be used without visible error. Spherical harmonics are in general only suitable for "low frequency" lighting effects, where the incident lighting distributions, visibility changes, and BSDFs are relatively blurry. In general, impractically large numbers of SH coefficients are needed to accurately represent higher frequency effects. Adaptive basis functions,



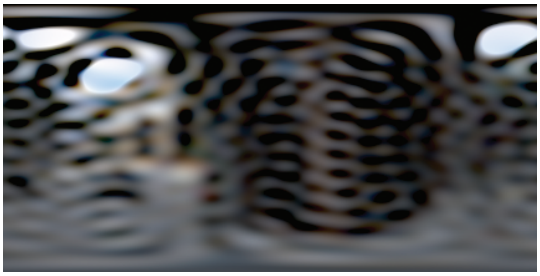
(a)



(b)



(c)



(d)

Figure 17.3: Environment Map Represented in the Spherical Harmonics Basis. (a) Original image, (b) Image represented with 5 SH bands, to $l_{\max} = 4$, (c) 10 SH bands, $l_{\max} = 9$, (d) 29 SH bands, $l_{\max} = 8$. Increasing the number of SH bands (and thus, coefficients) used to represent this complex spherical function improves the fidelity of the approximation. Even hundreds of SH coefficients can't represent the full complexity of this function, however.

such as nonlinear approximations based on wavelets, have been found to be better suited to "all frequency" lighting; see the "Further Reading" section for more details.

Functions for spherical harmonics are defined in the files `core/sh.h` and `core/sh.cpp`. Some additional functions related to rotating SH functions are defined in the file `core/shrots.cpp`; this file is quite large and so has been isolated from the main `sh.cpp` implementation file in order to reduce compile times when making changes to the `sh.cpp` file that don't involve the rotation functions.

17.2.1 EFFICIENT EVALUATION

It's useful to be able to efficiently evaluate the real SH functions up to arbitrary bands $l = l_{\max}$. For many real-time applications, it is worthwhile to precompute and tabularize the SH functions into cube map textures and then use fixed-function texture hardware to look up values from these cube maps. Here, we will develop a reasonably efficient SH evaluation routine that computes the basis function values for any given direction, though we will not use precomputed tables in the implementations below.

Two utility routines will be useful in the implementations of SH functions below. The first, `SHTerms()` gives the total number of basis functions (or equivalently, the total number of coefficients) that there are for a given $l_{\max}$ value. This function is used frequently for tasks like computing the size of arrays needed to hold SH coefficients.

⟨*Spherical Harmonics Declarations*⟩ ≡
```
inline int SHTerms(int lmax) {
    return (lmax + 1) * (lmax + 1);
}
```

Given a 1D array to be used for storing SH basis function coefficients, the `SHIndex()` computes the corresponding index into the array for a particular (l, m) basis function.

⟨*Spherical Harmonics Declarations*⟩ +≡
```
inline int SHIndex(int l, int m) {
    return l*l + l + m;
}
```

Now we can define the SH evaluation function, `SHEvaluate()`. `SHEvaluate()` takes a direction vector `w` and a maximum number of SH bands $l_{\max}$. It evaluates the real SH functions up to $l = l_{\max}$, storing the values of the functions in the provided `out` array, using the indexing scheme defined by `SHIndex()`. The length of `out` must be at least `SHTerms(lmax)`.

There are a few challenges in the implementation of this function. The first is efficiently bridging from the Cartesian (x, y, z) coordinates that `pbrt` uses for representing direction vectors to the spherical (θ, ϕ) coordinates that the real spherical harmonics are defined in terms of. The second will be applying a number of different recurrences to efficiently incrementally compute the associated Legendre polynomials and other required

SHIndex() 935

values. Using these recurrences is important: a naïve implementation based on evaluating each associated Legendre polynomials directly would be substantially less efficient.[1]

⟨*Spherical Harmonics Definitions*⟩ ≡
```
void SHEvaluate(const Vector &w, int lmax, float *out) {
    ⟨Compute Legendre polynomial values for cos θ 936⟩
    ⟨Compute K_l^m coefficients 939⟩
    ⟨Compute sin φ and cos φ values 940⟩
    ⟨Apply SH definitions to compute final (l, m) values 941⟩
}
```

The implementation starts by computing the associated Legendre polynomial values $P_l^m(\cos\theta)$, using the `legendrep()` utility routine. It stores the results in the `out` array; after these values have been stored, the implementation below will multiply by the remaining factors from Equation (17.7). Recall from Section 5.5.2 that in spherical coordinates, $\cos\theta = z$. Thus, we can just pass the z component of `w` in to the `legendrep()` function.

⟨*Compute Legendre polynomial values for* cos θ⟩ ≡ 936
```
legendrep(w.z, lmax, out);
```

There are four main steps in the implementation of `legendrep()`; each one uses a different recurrence to compute a different subset of the associated Legendre polynomial values for the given value. Figure 17.4 illustrates which subset of (l, m) values each step computes. The code here fills in the given `out` array where index `SHIndex(l,m)` holds the value of the associated Legendre polynomial for the given x value. It only computes the values of $P_l^m(x)$ for $m \geq 0$; the code in `SHEvaluate()` doesn't need the values for $m < 0$ in the end, so they aren't computed here.[2]

For the convenience of the code to follow, `legendrep()` starts by defining a macro `P()` that concisely indexes into the given output array.

⟨*Spherical Harmonics Local Definitions*⟩ ≡
```
static void legendrep(float x, int lmax, float *out) {
#define P(l,m) out[SHIndex(l,m)]
    ⟨Compute m = 0 Legendre values using recurrence 938⟩
    ⟨Compute m = l edge using Legendre recurrence 938⟩
    ⟨Compute m = l − 1 edge using Legendre recurrence 939⟩
    ⟨Compute m = 1, ..., l − 2 values using Legendre recurrence 939⟩
#undef P
}
```

First, the associated Legendre polynomials $P_l^0(x)$ are computed. The $m = 0$ values are the same as the regular Legendre polynomials $P_l(x)$. The first two of them are $P_0^0(x) = 1$

legendrep() 936
Vector 57

1 Even more efficient SH evaluation routines can be developed by writing a specialized code generator that generates C code to evaluate the recurrence relations directly.

2 There is commented-out code in the implementation to compute these values, using yet another recurrence, should those values be specifically needed for other applications.

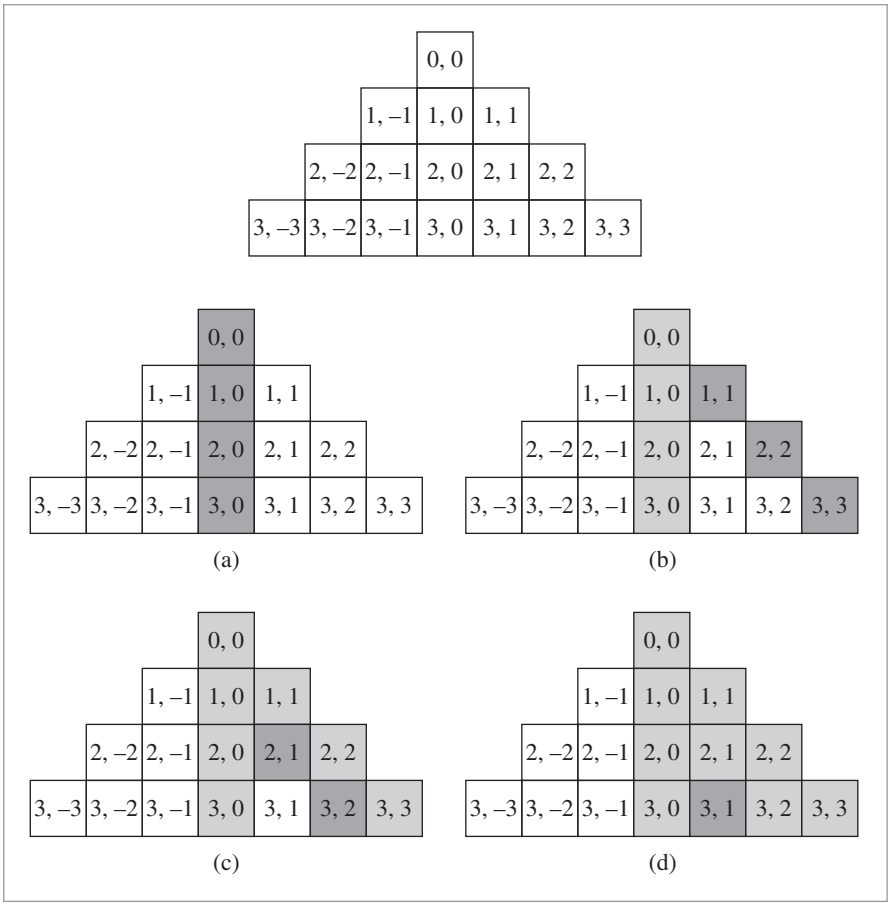


Figure 17.4: The Steps of Computing the Values of the Associated Legendre Polynomials. A series of recurrences is used to compute their values in the function `legendrep()`. At each step, the newly computed values have dark shading and the previously computed values have light shading. (a) The values along $m = 0$ are computed for all l bands, (b) the $m = l$ edge is computed from $l = 1$ to $l = l_{\max}$, (c) the $m = l - 1$ edge is computed for $l = 2$ to $l = l_{\max}$, (d) finally, remaining points $m = 1$ to $m = l - 2$ are computed for bands $l = 3$ and higher.

and $P_1^0(x) = x$. Subsequent ones are given by the recurrence

$$(l + 1)P_{l+1}(x) = (2l + 1)xP_l(x) - lP_{l-1}(x).$$

Rearranging terms and shifting the indexing by subtracting one from all of the l terms (which defines the same recurrence), we can see that if we want to compute the Legendre polynomial $P_l(x)$ and already know the values of the Legendre polynomials $P_{l-1}(x)$ and $P_{l-2}(x)$, then we have

$$P_l(x) = \frac{(2l - 1)xP_{l-1}(x) - (l - 1)P_{l-2}(x)}{l}.$$

Given this, the implementation is straightforward. (Note, however, that the loop indexing goes up to and includes $l = l_{\max}$.)

⟨*Compute $m = 0$ Legendre values using recurrence*⟩ ≡ 936
```
P(0,0) = 1.f;
P(1,0) = x;
for (int l = 2; l <= lmax; ++l)
    P(l, 0) = ((2*l-1)*x*P(l-1,0) - (l-1)*P(l-2,0)) / l;
```

After computing the $m = 0$ strip, the next step is to compute the values along the outer edge, where $m = l$. For these values we use following recurrence:

$$P_l^l(x) = -1^l(2l-1)!!(1-x^2)^{l/2}.$$

In this expression, !! denotes the double factorial, which is defined as

$$x!! = \begin{cases} 1 & x = 0 \text{ or } x = 1 \\ x(x-2)!! & \text{otherwise.} \end{cases}$$

The code in the fragment below incrementally computes this value for l values from 1 up to $l_{\max}$. The variable `neg` tracks the -1^l factor, `dfact` tracks the $(2l-1)!!$ factor, and `xpow` tracks the $(1-x^2)^{l/2}$ factor. These values are initialized with their appropriate values for $l = 1$, the initial condition going into the `for` loop.

After the value of $P_l^l(x)$ is updated in the loop, using the current values of these three variables, they are updated for the next pass through the loop. `neg` is negated, reflecting the effect of multiplication by another -1 factor, and the incremental effect of the next l value to the double factorial is incorporated in `dfact`. Finally, `xpow` is updated to account for the increment in l, which corresponds to an additional factor of $(1-x^2)^{1/2}$ using the value `xroot`, which stores this value.

⟨*Compute $m = l$ edge using Legendre recurrence*⟩ ≡ 936
```
float neg = -1.f;
float dfact = 1.f;
float xroot = sqrtf(max(0.f, 1.f - x*x));
float xpow = xroot;
for (int l = 1; l <= lmax; ++l) {
    P(l, l) = neg * dfact * xpow;
    neg *= -1.f;      // neg = (-1)^l
    dfact *= 2*l + 1; // dfact = (2*l-1)!!
    xpow *= xroot;    // xpow = powf(1.f - x*x, float(l) * 0.5f);
}
```

Next, the slice where $m = l - 1$ is computed. These values are found using the recurrence

$$P_{l+1}^l(x) = x(2l+1)P_l^l(x),$$

which is equivalent to

$$P_l^{l-1}(x) = x(2l-1)P_{l-1}^{l-1}(x),$$

if l is shifted by -1. Since we have computed $P_1^1(x)$ in the previous code, this loop starts with $l = 2$ and implements this recurrence directly.

⟨*Compute $m = l - 1$ edge using Legendre recurrence*⟩ ≡ 936
```
for (int l = 2; l <= lmax; ++l)
    P(l, l-1) = x * (2*l-1) * P(l-1, l-1);
```

We can now finally fill in the remainder of the values for $0 < m < l - 1$ using one last recurrence,

$$(l - m + 1)P_{l+1}^m(x) = (2l + 1)xP_l^m(x) - (l + m)P_{l-1}^m(x).$$

Once again we offset l by -1 and divide the right-hand side by the resulting $(l - m)$ factor in the implementation below to compute the value of $P_l^m(x)$. Note that the loop starting and ending points are chosen carefully so that they don't recompute values that the previous recurrences have already covered.

⟨*Compute $m = 1, \ldots, l - 2$ values using Legendre recurrence*⟩ ≡ 936
```
for (int l = 3; l <= lmax; ++l)
    for (int m = 1; m <= l-2; ++m)
        P(l, m) = ((2 * (l-1) + 1) * x * P(l-1,m) -
                   (l-1+m) * P(l-2,m)) / (l - m);
```

Next, the `SHEvaluate()` function needs to compute the K_l^m coefficients using Equation (17.6). A separate temporary array is allocated to store them here. Because the K_l^m values are fixed for given (l, m) values and because code that uses the spherical harmonics routines will generally only go up to a fixed $l = l_{\text{max}}$ value, a more optimized implementation might require the caller to compute an array holding these values once at startup time and then pass them into the SH routines. The implementation here doesn't impose this requirement, in the interests of simpler interfaces for the SH routines.[3]

⟨*Compute K_l^m coefficients*⟩ ≡ 936
```
float *Klm = ALLOCA(float, SHTerms(lmax));
for (int l = 0; l <= lmax; ++l)
    for (int m = -l; m <= l; ++m)
        Klm[SHIndex(l, m)] = K(l, m);
```

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static inline float K(int l, int m) {
    return sqrtf((2.f * l + 1.f) * INV_FOURPI * divfact(l, m));
}
```

The `divfact()` function efficiently computes the $(l - |m|)!/(l + |m|)$ term of the K_l^m function using the relationship

$$\frac{(a - |b|)!}{(a + |b|)!} = \frac{1}{(a - |b| + 1)(a - |b| + 2)\ldots(a + |b|)}$$

ALLOCA() 1009
divfact() 940
INV_FOURPI 1002
K() 939
SHIndex() 935
SHTerms() 935

3 A third alternative—probably the best—would be to have a local array initialized with precomputed K_l^m coefficients up to a large l_{max} value.

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static inline float divfact(int a, int b) {
    if (b == 0) return 1.f;
    float fa = a, fb = fabsf(b);
    float v = 1.f;
    for (float x = fa-fb+1.f; x <= fa+fb; x += 1.f)
        v *= x;
    return 1.f / v;
}
```

`SHEvaluate()` next computes the sine and cosine factors of Equation (17.7). From this equation, the real spherical harmonics have factors of the form $\sin(-m\phi)$ and $\cos(m\phi)$; thus, we need the values of $\sin -\phi$, $\sin(-2\phi)$, and so forth up to $\sin(-m\phi)$ for $m = l_{\max}$ and similarly for the cosine factors.

Space is allocated for the sine and cosine values in the `sins` and `coss` arrays. When these are filled in, the element `sins[i]` will hold the value $\sin(i\phi)$ and so forth. We use the relationships from page 425 to start this process; $\sin\phi = y/\sqrt{1-z^2}$ and $\cos\phi = x/\sqrt{1-z^2}$. If z is nearly one or negative one so that $\sqrt{1-z^2}$ is zero, then the direction is essentially toward the $z = 1$ or $z = -1$ pole and the sine and cosine values are set to arbitrary (but mutually consistent) values. Otherwise, the `sinCosIndexed()` routine computes all of the sine and cosine factors, given the values of the first ones computed and passed in here.

⟨*Compute* $\sin\phi$ *and* $\cos\phi$ *values*⟩ ≡ 936
```
float *sins = ALLOCA(float, lmax+1), *coss = ALLOCA(float, lmax+1);
float xyLen = sqrtf(max(0.f, 1.f - w.z*w.z));
if (xyLen == 0.f) {
    for (int i = 0; i <= lmax; ++i) sins[i] = 0.f;
    for (int i = 0; i <= lmax; ++i) coss[i] = 1.f;
}
else
    sinCosIndexed(w.y / xyLen, w.x / xyLen, lmax+1, sins, coss);
```

There are recurrence formulas that can be used to give the values of $\sin(i\phi)$ and $\cos(i\phi)$ in terms of $\sin((i-1)\phi)$ and $\cos((i-1)\phi)$:

$$\sin(x+y) = \sin x \cos y + \cos x \sin y$$
$$\cos(x+y) = \cos x \cos y - \sin x \sin y$$

The direct implementation of these, for $x = y = \phi$ and where $\sin\phi$ and $\cos\phi$ are passed in as parameters, is in the code below.

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static void sinCosIndexed(float s, float c, int n,
                          float *sout, float *cout) {
    float si = 0, ci = 1;
    for (int i = 0; i < n; ++i) {
        ⟨Compute sin iφ and cos iφ using recurrence 941⟩
    }
}
```

ALLOCA() 1009
sinCosIndexed() 940

⟨*Compute* sin $i\phi$ *and* cos $i\phi$ *using recurrence*⟩ ≡ 940
```
*sout++ = si;
*cout++ = ci;
float oldsi = si;
si = si * c + ci * s;
ci = ci * c - oldsi * s;
```

Given all of these components, the `SHEvaluate()` function can now finish computing the values from Equation (17.7). Going into this fragment, the `out` array holds the values of the associated Legendre polynomials for the corresponding (l, m) indices. For each term, then, we need to multiply in the K_l^m normalization factor and the sine or cosine factor. The implementation below uses the fact that $\sin -x = -\sin x$. For the $m < 0$ terms, note that the associated Legendre polynomial values for $m > 0$ are used as per the definition in Equation (17.7).

⟨*Apply SH definitions to compute final* (l, m) *values*⟩ ≡ 936
```
static const float sqrt2 = sqrtf(2.f);
for (int l = 0; l <= lmax; ++l) {
    for (int m = -l; m < 0; ++m)
        out[SHIndex(l, m)] = sqrt2 * Klm[SHIndex(l, m)] *
            out[SHIndex(l, -m)] * sins[-m];
    out[SHIndex(l, 0)] *= Klm[SHIndex(l, 0)];
    for (int m = 1; m <= l; ++m)
        out[SHIndex(l, m)] *= sqrt2 * Klm[SHIndex(l, m)] * coss[m];
}
```

17.2.2 PROJECTING LIGHT SOURCES

Now that we have routines for evaluating the spherical harmonics functions, we will start to add infrastructure to other parts of `pbrt` to support them. Given a light source and a particular point p in the scene, the light source defines a directionally varying incident radiance function $L_i(\mathrm{p}, \omega)$ at the point. For point light sources, the incident radiance comes from only a single direction, but for area or infinite area lights, many directions may carry incident illumination. In this section, we will add methods to the various light sources to project their incident radiance functions into SH coefficients.

The `Light::SHProject()` method computes the SH coefficients that represent the incident radiance function due to the light source in the SH basis up to the band $l = l_{\max}$. In this case, the coefficients aren't just scalars but are spectra; they are returned in the array `coeffs`. Here, we will define specific implementations of this method for point and infinite area light sources as well as a general implementation that uses quasi-Monte Carlo integration to compute SH coefficients at a point for the incident radiance function from arbitrary light sources.

Point 63
RNG 1003
Scene 22
SHIndex() 935
Spectrum 263

⟨*Light Interface*⟩ +≡ 606
```
virtual void SHProject(const Point &p, float pEpsilon, int lmax,
    const Scene *scene, bool computeLightVisibility, float time,
    RNG &rng, Spectrum *coeffs) const;
```

Point Lights

After initializing the coefficients to zero, the `PointLight::SHProject()` method checks to see if the point light source is visible from the given point. If not, the incident radiance is zero and its work is done. Note that the user can indicate whether occlusion of geometric objects in the scene should be ignored or not. For some applications, it's useful to use the unoccluded incident radiance function; for others, the effect of shadowing by geometry should be incorporated.

⟨*PointLight Method Definitions*⟩ +≡

```
void PointLight::SHProject(const Point &p, float pEpsilon, int lmax,
        const Scene *scene, bool computeLightVisibility, float time,
        RNG &rng, Spectrum *coeffs) const {
    for (int i = 0; i < SHTerms(lmax); ++i)
        coeffs[i] = 0.f;
    if (computeLightVisibility &&
        scene->IntersectP(Ray(p, Normalize(lightPos - p), pEpsilon,
                              Distance(lightPos, p), time)))
        return;
    ⟨Project point light source to SH 942⟩
}
```

Because point light sources are defined by a delta distribution, projecting their contribution into SH with Equation (17.8) is straightforward. If we define ω_i to be the normalized direction from the point p to the point light source position $\mathrm{p_l}$ and take advantage of the fact that the point light is defined by a delta distribution (recall Section 14.6.2), then the integral turns into a single term to evaluate:

$$\begin{aligned} c_l^m &= \int_{\mathbb{S}^2} L_\mathrm{i}(\mathrm{p}, \omega) Y_l^m(\omega)\, \mathrm{d}\omega \\ &= \int_{\mathbb{S}^2} \delta(\omega - \omega_\mathrm{i}) \frac{I}{||\mathrm{p_l} - \mathrm{p}||} Y_l^m(\omega)\, \mathrm{d}\omega \\ &= \frac{I}{||\mathrm{p_l} - \mathrm{p}||} Y_l^m(\omega_\mathrm{i}). \end{aligned}$$

⟨*Project point light source to SH*⟩ ≡ 942

```
float *Ylm = ALLOCA(float, SHTerms(lmax));
Vector wi = Normalize(lightPos - p);
SHEvaluate(wi, lmax, Ylm);
Spectrum Li = Intensity / DistanceSquared(lightPos, p);
for (int i = 0; i < SHTerms(lmax); ++i)
    coeffs[i] = Li * Ylm[i];
```

General Light Sources

For general light sources without specialized implementations of the method, the default implementation of `Light::SHProject()` uses quasi-Monte Carlo integration to estimate the SH coefficient values. The implementation here starts by computing random scramble values for generating low-discrepancy samples using the routines from Section 7.4.3

ALLOCA() 1009
Distance() 65
DistanceSquared() 65
Point 63
PointLight::Intensity 611
PointLight::lightPos 611
Ray 66
RNG 1003
Scene 22
Scene::IntersectP() 24
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Vector 57
Vector::Normalize() 63

and allocating a temporary array, `Ylm`, for holding values of the SH basis functions returned by `SHEvaluate()`.

⟨*Light Method Definitions*⟩ +≡
```
void Light::SHProject(const Point &p, float pEpsilon, int lmax,
        const Scene *scene, bool computeLightVisibility, float time,
        RNG &rng, Spectrum *coeffs) const {
    for (int i = 0; i < SHTerms(lmax); ++i)
        coeffs[i] = 0.f;
    uint32_t ns = RoundUpPow2(nSamples);
    uint32_t scramble1D = rng.RandomUInt();
    uint32_t scramble2D[2] = { rng.RandomUInt(), rng.RandomUInt() };
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (uint32_t i = 0; i < ns; ++i) {
        ⟨Compute incident radiance sample from light, update SH coeffs 943⟩
    }
}
```

For each sample to be taken of the estimator, a `LightSample` is created to sample the light source.

⟨*Compute incident radiance sample from* `light`*, update SH* `coeffs`⟩ ≡ 943
```
float u[2], pdf;
Sample02(i, scramble2D, u);
LightSample lightSample(u[0], u[1], VanDerCorput(i, scramble1D));
Vector wi;
VisibilityTester vis;
Spectrum Li = Sample_L(p, pEpsilon, lightSample, time, &wi, &pdf, &vis);
if (!Li.IsBlack() && pdf > 0.f &&
    (!computeLightVisibility || vis.Unoccluded(scene))) {
    ⟨Add light sample contribution to MC estimate of SH coefficients 943⟩
}
```

The values of the coefficients are found using Equation (17.11).

⟨*Add light sample contribution to MC estimate of SH coefficients*⟩ ≡ 943
```
SHEvaluate(wi, lmax, Ylm);
for (int j = 0; j < SHTerms(lmax); ++j)
    coeffs[j] += Li * Ylm[j] / (pdf * ns);
```

ALLOCA() 1009
InfiniteAreaLight 629
Light::Sample_L() 608
LightSample 710
Point 63
RNG 1003
RNG::RandomUInt() 1003
RoundUpPow2() 1002
Sample02() 372
Scene 22
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Spectrum::IsBlack() 265
VanDerCorput() 372
Vector 57
VisibilityTester 608
VisibilityTester::
 Unoccluded() 609

Infinite Area Lights

Recall that `InfiniteAreaLights` are represented by 2D texture maps that define emitted radiance as a function of (θ, ϕ) incident directions in spherical coordinates. Its `SHProject()` method here uses one of three different approaches, based on whether visibility from occluding objects should be included in the incident radiance function and based on the resolution of the image map.

⟨*InfiniteAreaLight Method Definitions*⟩ +≡
```
void InfiniteAreaLight::SHProject(const Point &p, float pEpsilon,
        int lmax, const Scene *scene, bool computeLightVis,
        float time, RNG &rng, Spectrum *coeffs) const {
    ⟨Project InfiniteAreaLight to SH using Monte Carlo if visibility needed 944⟩
    for (int i = 0; i < SHTerms(lmax); ++i)
        coeffs[i] = 0.f;
    int ntheta = radianceMap->Height(), nphi = radianceMap->Width();
    if (min(ntheta, nphi) > 50) {
        ⟨Project InfiniteAreaLight to SH from lat-long representation 945⟩
    }
    else {
        ⟨Project InfiniteAreaLight to SH from cube map sampling 949⟩
    }
}
```

If occlusion from objects in the scene should be included in the coefficients, Monte Carlo integration is a reasonable approach; although the other two methods for `InfiniteAreaLights` can be modified to handle this case, it's most efficiently handled here.

⟨*Project* `InfiniteAreaLight` *to SH using Monte Carlo if visibility needed*⟩ ≡ 944
```
if (computeLightVis) {
    Light::SHProject(p, pEpsilon, lmax, scene, computeLightVis,
                     time, rng, coeffs);
    return;
}
```

As an alternative to Monte Carlo integration, the incident radiance function from an infinite area light can be projected into the SH basis using Riemann integration, based on looping over all of the texels in the environment map. Relatively low-resolution environment maps are handled using a slightly different method that gives better accuracy in that case.

We can equivalently express the SH projection as integral over (θ, ϕ) directions, versus the definition over directions ω in Equation (17.8). Recall from Equation (5.4) that an extra $\sin\theta$ factor is introduced when transforming to an integral over (θ, ϕ), giving

$$c_l^m = \int_0^{2\pi} \int_0^{\pi} L_{\mathrm{i}}(\theta, \phi)\, Y_l^m(\theta, \phi) \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi.$$

We can then compute an approximation to this integral as a 2D Riemann sum over the elements of the environment map,

$$c_l^m \approx \frac{\pi}{N_\theta} \sum_t^{N_\theta} \frac{2\pi}{N_\phi} \sum_p^{N_\phi} L_{\mathrm{i}}(\theta_t, \phi_p)\, Y_l^m(\theta_t, \phi_p) \sin\theta_t, \qquad [17.12]$$

where (θ_t, ϕ_p) is the (θ, ϕ) direction corresponding to the environment map texel at (t, p) (recall Section 5.5.2).

Computing this sum involves looping over all of the texels and applying the formula above. The fragment ⟨*Precompute theta and phi values for lat-long map projection*⟩ (not

InfiniteAreaLight 629
InfiniteAreaLight::
radianceMap 631
Light::SHProject() 943
MIPMap::Height() 535
MIPMap::Width() 535
Point 63
RNG 1003
Scene 22
SHTerms() 935
Spectrum 263

included here) fills in the arrays sintheta, costheta, sinphi, and cosphi, of sizes ntheta and nphi such that sintheta[i] corresponds to the value of sin θ for the ith texel in the theta direction and so forth.

⟨*Project* InfiniteAreaLight *to SH from lat-long representation*⟩ ≡ 944
```
⟨Precompute θ and φ values for lat-long map projection⟩
float *Ylm = ALLOCA(float, SHTerms(lmax));
for (int theta = 0; theta < ntheta; ++theta) {
    for (int phi = 0; phi < nphi; ++phi) {
        ⟨Add InfiniteAreaLight texel's contribution to SH coefficients 945⟩
    }
}
⟨Free memory used for lat-long theta and phi values⟩
```

For each texel, we compute the world-space direction vector ω that corresponds to its incident illumination direction. The running coeffs sum can then be updated with the contribution given by Equation (17.12).

⟨*Add* InfiniteAreaLight *texel's contribution to SH coefficients*⟩ ≡ 945
```
Vector w = Vector(sintheta[theta] * cosphi[phi],
                  sintheta[theta] * sinphi[phi],
                  costheta[theta]);
w = Normalize(LightToWorld(w));
Spectrum Le = Spectrum(radianceMap->Texel(0, phi, theta),
                       SPECTRUM_ILLUMINANT);
SHEvaluate(w, lmax, Ylm);
for (int i = 0; i < SHTerms(lmax); ++i)
    coeffs[i] += Le * Ylm[i] * sintheta[theta] *
        (M_PI / ntheta) * (2.f * M_PI / nphi);
```

For very-low-resolution environment maps, the Riemann approach can suffer from noticeable numeric error. Consider as an extreme the case of a single-texel environment map. This environment map represents uniform illumination from all directions, yet the Riemann sum will compute coefficients that are essentially the same as those for a distant light illuminating the receiving point from a single direction. The problem is that the factors of the sum other than the incident radiance function—the $Y_l^m(\theta, \phi)$ functions and the sin θ factor—aren't being sampled at a high enough rate to capture enough of their variation to give a good result.

One option would be to resample the environment map to a higher resolution in this case, either explicitly or by taking more Riemann samples than there are texels. However, we will instead use a different approach based on the SHProjectCube() function, which projects a directionally varying function specified as a *cube map* into the SH basis. We will discuss the implementation of this function now before showing its use for InfiniteAreaLights.

Cube maps define a directionally varying function with six square two-dimensional texture maps, one along each of the $\pm x$, $\pm y$, and $\pm z$ coordinate axes. Cube maps are usually specified with their faces at unit distance from the origin. For example, consider the $-x$ face of a cube map at $x = -1$. The y and z coordinates of this face range from $[-1, 1]$.

ALLOCA() 1009
InfiniteAreaLight::
radianceMap 631
Light::LightToWorld 606
MIPMap::Texel() 535
M_PI 1002
SHEvaluate() 936
SHProjectCube() 946
SHTerms() 935
Spectrum 263
SPECTRUM_ILLUMINANT 277
Vector 57
Vector::Normalize() 63

Thus, given a texel at particular (y, z) coordinates on the $-x$ axis, the corresponding direction vector is $(-1, y, z)$. The ability to project functions defined by cube map faces into SH is useful in a number of other situations, most notably for environment maps already represented as cube maps. Rendering systems based on rasterization can more easily generate a cubical environment map than other parameterizations of the unit sphere of directions.

`SHProjectCube()` is a template function, parameterized by a function `func` that is called to sample the value of the underlying function in a particular direction. `func` should have the signature

```
Spectrum func(int u, int v, const Point &p, const Vector &w);
```

The `u` and `v` values passed to the callback function will be texel coordinates in the range [0, `res`), where `res` is the sampling resolution passed to `SHProjectCube()`, `p` is the point passed in, and `w` is the non-normalized direction vector from the origin to the point on the face.

The callback function can either be a class or `struct` with an `operator()` method, as the `InfiniteAreaCube` structure does below, or it may be a pointer to a regular function. Using an `operator()` is generally more efficient, since the callback can be expanded inline into the instantiation of `SHProjectCube()` if the method is defined inline to the class or structure.

The implementation of `SHProjectCube()` also does a Riemann sum for each face of the cube; in the general 2D setting, it is

$$\int_{x_a}^{x_b}\int_{y_a}^{y_b} f(x, y)\,\mathrm{d}x\,\mathrm{d}y \approx \frac{x_b - x_a}{N_x}\sum_i \frac{y_b - y_a}{N_y}\sum_j f(x_i, y_j).$$

The implementation of `SHProjectCube()` loops over the samples of a canonical face and updates the Riemann sum's contribution for all six faces of the cube map in each iteration.

⟨*Spherical Harmonics Declarations*⟩ +≡

```
template <typename Func>
void SHProjectCube(Func func, const Point &p, int res, int lmax,
                   Spectrum *coeffs) {
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (int u = 0; u < res; ++u) {
        float fu = -1.f + 2.f * (float(u) + 0.5f) / float(res);
        for (int v = 0; v < res; ++v) {
            float fv = -1.f + 2.f * (float(v) + 0.5f) / float(res);
            ⟨Incorporate results from +z face to coefficients 948⟩
            ⟨Incorporate results from other faces to coefficients⟩
        }
    }
}
```

ALLOCA() 1009
Point 63
SHTerms() 935
Spectrum 263

This task is made slightly more complex by the fact that we actually want to compute the integral of a function of directions on the unit sphere defined by the faces of a cube map,

rather than a simple 2D integral over a planar region. As such, a corrective factor has to be added to account for the distortion from the mapping from the planar faces of the cube map to the surface of the sphere. Intuitively, texels toward the edges of the cube map faces project to a smaller area on the unit sphere than those toward the middle. This factor is found by expressing the change of variables that corresponds to this situation and then computing the Jacobian of the resulting transformation function. In the general n-dimensional case, given a mapping between two domains $(y_1, \ldots, y_n) = g(x_1, \ldots, x_n)$, we have

$$\int f(x_1, \ldots, x_n)\, \mathrm{d}x_1 \ldots \mathrm{d}x_n = \int f(y_1, \ldots, y_n) \left| \frac{\partial(x_1, \ldots, x_n)}{\partial(y_1, \ldots, y_n)} \right| \mathrm{d}y_1 \ldots \mathrm{d}y_n,$$

where the partial derivatives term is the Jacobian of the mapping.

As a concrete example, consider integrating over the $+z$ face of a cube map. Based on the cube map definition, we can see see that the function g that maps points (x, y) on the $z = 1$ face to points on a sphere of radius r (x', y', z') is

$$(x', y', z') = g(x, y, r) = \left(\frac{rx}{\sqrt{x^2 + y^2 + 1}}, \frac{ry}{\sqrt{x^2 + y^2 + 1}}, \frac{r}{\sqrt{x^2 + y^2 + 1}}, \right).$$

Finding the entries of the Jacobian matrix and then the matrix determinant is a straightforward algebraic exercise; the end result is:

$$\frac{1}{(1 + x^2 + y^2)^{3/2}}.$$

This is the term that corrects for the varying density of cube map samples when mapped to the unit sphere. Because z was assumed to be 1 in the derivation above, more generally we can see that an appropriate correction term that can be used for all faces is:

$$\frac{1}{(x^2 + y^2 + z^2)^{3/2}}.$$

There is also an elegant geometric derivation of the change of variables term. Recall from Section 5.5.2 the relationship between differential area and projected solid angle,

$$\mathrm{d}\omega = \frac{\mathrm{d}A \cos\theta}{r^2}.$$

Given a differential area $\mathrm{d}A$ at a coordinate (x, y) on the $+z$ face of the cube map, it's easy to see that $r = \sqrt{x^2 + y^2 + 1}$ and thus $r^2 = x^2 + y^2 + 1$. We can also take advantage of the fact that the cosine of an angle in a right triangle is the ratio of the adjacent side to the hypotenuse: in Figure 17.5, note that the hypotenuse is r and the adjacent side has length 1. Therefore, $\cos\theta = 1/r$ and

$$\mathrm{d}\omega = \frac{\mathrm{d}A(1/r)}{r^2} = \frac{\mathrm{d}A}{r^3} = \frac{\mathrm{d}A}{(x^2 + y^2 + z^2)^{3/2}}.$$

Given all this work, computing the term for a sample on a face is straightforward. Here is the fragment for the $+z$ face; the others are similar and not included here. Recall that the integral must be computed independently for each of the `SHTerms(lmax)` coefficients; the final loop over `k` updates each of these.

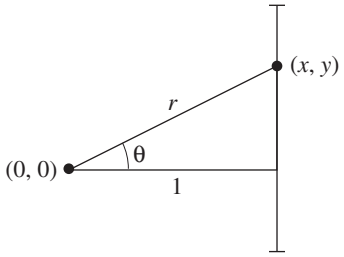


Figure 17.5: Geometric Setting for Computing the Relationship Between Differential Area on the Cube Map and on the Unit Sphere. The cosine of the angle θ is given by the ratio of the length of the adjacent side of the triangle, 1, and the hypotenuse, r.

⟨*Incorporate results from +z face to coefficients*⟩ ≡ 946
```
Vector w(fu, fv, 1);
SHEvaluate(Normalize(w), lmax, Ylm);
Spectrum f = func(u, v, p, w);
float dA = 1.f / powf(Dot(w, w), 3.f/2.f);
for (int k = 0; k < SHTerms(lmax); ++k)
    coeffs[k] += f * Ylm[k] * dA * (4.f / (res * res));
```

To use the `SHProjectCube()` function with infinite area lights, the `InfiniteAreaCube` structure is defined; it returns the appropriate color for the given direction from the infinite area light source environment map. Note that we don't create an explicit representation of the environment map in the form of a cube map here, but just compute the appropriate color for the sampled directions on cube map faces generated by `SHProjectCube()`.

⟨*InfiniteAreaLight Utility Classes*⟩ ≡
```
struct InfiniteAreaCube {
    ⟨InfiniteAreaCube Public Methods 948⟩
    const InfiniteAreaLight *light;
    const Scene *scene;
    float time, pEpsilon;
    bool computeVis;
};
```

⟨*InfiniteAreaCube Public Methods*⟩ ≡ 948
```
InfiniteAreaCube(const InfiniteAreaLight *l, const Scene *s,
                 float t, bool cv, float pe)
    : light(l), scene(s), time(t), pEpsilon(pe), computeVis(cv) { }
```

When the callback is called, the infinite light source's color is easily determined using existing methods.

InfiniteAreaCube 948
InfiniteAreaLight 629
Scene 22
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Vector 57
Vector::Normalize() 63

⟨*InfiniteAreaCube Public Methods*⟩ +≡ 948
```
Spectrum operator()(int, int, const Point &p, const Vector &w) {
    Ray ray(p, w, pEpsilon, INFINITY, time);
    if (!computeVis || !scene->IntersectP(ray))
        return light->Le(RayDifferential(ray));
    return 0.f;
}
```

For projecting low-resolution latitude-longitude maps into SH, a sampling resolution of 200×200 on each of the cube map faces is used, giving good directional sampling even if the original environment map is low resolution.

⟨*Project* `InfiniteAreaLight` *to SH from cube map sampling*⟩ ≡ 944
```
SHProjectCube(InfiniteAreaCube(this, scene, time, computeLightVis,
                               pEpsilon),
              p, 200, lmax, coeffs);
```

17.2.3 PROJECTING INCIDENT RADIANCE FUNCTIONS

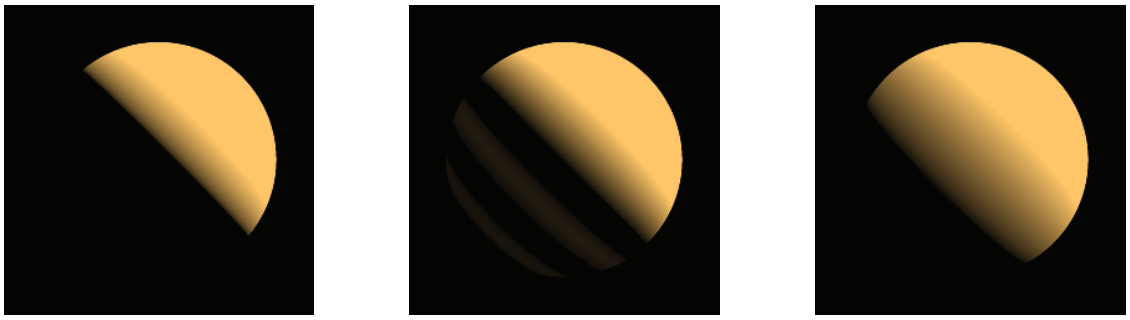
`pbrt` provides two utility functions for computing the complete incident radiance function at a point and projecting the function to the spherical harmonic basis. The first of them computes direct lighting from all light sources; the second computes direct and indirect lighting. Both will be used in the next section for the `CreateRadianceProbes` `Renderer`.

`SHProjectIncidentDirectRadiance()` loops over all of the lights in the scene and accumulates the sum of their SH coefficients into the `c_d` output array. Its only unusual feature is the call to `SHReduceRinging()`; this function (which will be introduced shortly) reduces some of the artifacts that can arise from projecting high-frequency functions into the SH basis when, as is commonly the case, there aren't enough SH coefficients to perfectly reconstruct the original function.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHProjectIncidentDirectRadiance(const Point &p, float pEpsilon,
        float time, MemoryArena &arena, const Scene *scene,
        bool computeLightVis, int lmax, RNG &rng, Spectrum *c_d) {
    ⟨Loop over light sources and sum their SH coefficients 949⟩
    SHReduceRinging(c_d, lmax);
}
```

⟨*Loop over light sources and sum their SH coefficients*⟩ ≡ 949
```
Spectrum *c = arena.Alloc<Spectrum>(SHTerms(lmax));
for (uint32_t i = 0; i < scene->lights.size(); ++i) {
    Light *light = scene->lights[i];
    light->SHProject(p, pEpsilon, lmax, scene, computeLightVis, time,
                     rng, c);
    for (int j = 0; j < SHTerms(lmax); ++j)
        c_d[j] += c[j];
}
```

CreateRadianceProbes 958
InfiniteAreaCube 948
InfiniteAreaCube::computeVis 948
InfiniteAreaCube::light 948
InfiniteAreaCube::pEpsilon 948
InfiniteAreaCube::scene 948
InfiniteAreaCube::time 948
INFINITY 1002
Light 606
Light::Le() 631
Light::SHProject() 943
MemoryArena 1015
Point 63
Ray 66
RayDifferential 69
Renderer 24
RNG 1003
Scene 22
Scene::IntersectP() 24
SHProjectCube() 946
SHReduceRinging() 951
Spectrum 263
Vector 57



(a) (b) (c)

Figure 17.6: Ringing From Projecting a Function with High Frequencies into the SH Basis. (a) Sphere illuminated with standard direct lighting algorithms from a point light source. (b) When projected into SH, the point source's infinite frequency content causes ringing in the function in the basis, leading to shading errors. (c) If the coefficients are scaled using the `SHReduceRinging()` function, the result is substantially improved.

We won't include the implementation of `SHProjectIncidentIndirectRadiance()` here; it computes a Monte Carlo estimate of Equation (17.11), computing incident radiance with the `Li()` method of the `Renderer` passed into it.

⟨*Spherical Harmonics Declarations*⟩ +≡

```
void SHProjectIncidentIndirectRadiance(const Point &p, float pEpsilon,
    float time, const Renderer *renderer, Sample *origSample,
    const Scene *scene, int lmax, RNG &rng, int nSamples, Spectrum *c_i);
```

17.2.4 REDUCING RINGING

When projecting signals with high-frequency directional variation over the unit sphere (such as the incident radiance function from a point light or a small area light), an artifact that often occurs is *ringing* in the function that is reconstructed from the coefficients and the SH basis functions. This is exactly the same problem we encountered when using perfect reconstruction with sinc functions to reconstruct signals with discontinuities in Section 7.1.6.

Figure 17.6 shows the effect of ringing with spherical harmonics. On the left is a sphere illuminated by a point light source rendered using the `DirectLightingIntegrator`. In the middle is the result of the usual approach of projecting the incident radiance function for the point light into SH and then using it to compute illumination arriving at the sphere. The right shows the effect of implementing the technique described here.

One way to solve this problem is to prefilter (i.e., blur) the signal before projection into SH, so that it doesn't have any excessively high-frequency information in it. A simpler approach, which we will implement here, was suggested by Sloan (2008). His observation was that we don't necessarily want to strictly minimize the least-squares error of the projection of a function into the SH basis, but that it's also worthwhile to have an error metric that penalizes oscillations in the result.

DirectLightingIntegrator 742
Point 63
Renderer 24
RNG 1003
Sample 343
Scene 22
SHReduceRinging() 951
Spectrum 263

Sloan derived an approach to minimize the Laplacian of the reconstructed function, which reduces these oscillations. The result gives a scale factor for each coefficient that depends on a scaling factor λ and the coefficient's band l:

$$\tilde{c}_l^m = \frac{c_l^m}{1 + \lambda l^2 (l+1)^2}. \quad [17.13]$$

Note that if $\lambda = 0$ then this is the same as regular least-squares projection. Small values of λ generally work well; the default value of .005 was used in Figure 17.6(c). See Sloan's report for details on the derivation as well as more discussion about how to compute λ values.

The `SHReduceRinging()` function applies Equation (17.13) to a given set of SH coefficients.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHReduceRinging(Spectrum *c, int lmax, float lambda) {
    for (int l = 0; l <= lmax; ++l) {
        float scale = 1.f / (1.f + lambda * l * l * (l + 1) * (l + 1));
        for (int m = -l; m <= l; ++m)
            c[SHIndex(l, m)] *= scale;
    }
}
```

17.2.5 ROTATIONS

Many precomputed light transport algorithms need to rotate functions in spherical harmonics; for example, if both illumination and a reflection function are represented in the SH basis but illumination is represented in world-space coordinates and reflection is represented in local tangent-frame coordinates, then one of the two must be rotated to be in the same coordinate system as the other if reflected light is to be computed.

A very useful property of the spherical harmonic basis for rendering is that it is *rotationally invariant*; this means that, given some function $f(\omega)$ defined on the sphere with SH coefficients c_i, if one wants to compute the new SH coefficients that result from rotating the original function on the sphere, they can be computed directly from the original coefficients c_i and a SH rotation matrix $\mathbf{S}$:

$$c_i' = \mathbf{S} c_i.$$

Using a SH rotation matrix in this manner gives the same result as if the original function had been rotated and then projected into the SH basis. More specifically, no error is introduced by rotating with the coefficients directly versus rotating the original function and reprojecting into spherical harmonics.

The fact that coefficients for the rotated function can be computed directly from the coefficients of the original function is an extremely useful one for rendering for two reasons. First, computing the new coefficients c_i' from the original ones is generally much more efficient than rotating the original function and then re-computing the coefficients. Second, because no error is introduced in the rotation, visual artifacts aren't introduced in animations where objects are moving or rotating and SH functions involved in rendering them need to be rotated.

SHIndex() 935
Spectrum 263

Although there are closed-form expressions for the SH rotation matrices **S** that correspond to a given 3×3 coordinate-system rotation matrix **R**, these matrices are dense, complex, and computationally inefficient.[4] More efficient implementations are possible by decomposing the given rotation matrix into Euler angles and successively applying matrices that rotate the SH coefficients around individual coordinate axes. This section will develop this approach.

The basic setting is that we want to find new coefficients c_i' given the rotation matrix **R** and the original coefficients c_i. One way to derive this transformation is to use c_i to reconstruct the function in 3D, apply the 3×3 rotation matrix to rotate the function, and then project the resulting function back into SH. This is perhaps best understood with the $l = 1$ case. (When rotating functions in the SH basis, coefficients within any given l band don't affect coefficients in other bands—this can be shown from orthonormality. Therefore, we can consider the rotation of each band independently.)

We have coefficients c_1^{-1}, c_1^0, and c_1^1; the unrotated function in the SH basis is

$$\hat{f}(x, y, z) = c_1^{-1} Y_1^{-1}(x, y, z) + c_1^0 Y_1^0(x, y, z) + c_1^1 Y_1^1(x, y, z).$$

Using the Cartesian form of the basis functions in Equation (17.7), we have

$$\hat{f}(x, y, z) = c_1^{-1}\left(-\sqrt{\frac{3}{4\pi}}y\right) + c_1^0\left(-\sqrt{\frac{3}{4\pi}}z\right) + c_1^1\left(-\sqrt{\frac{3}{4\pi}}x\right).$$

We can rotate this function with the 3×3 rotation matrix **R**. Define the rotated coordinates as usual by

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} r_{0,0} & r_{0,1} & r_{0,2} \\ r_{1,0} & r_{1,1} & r_{1,2} \\ r_{2,0} & r_{2,1} & r_{2,2} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix},$$

and then the rotated version of $\hat{f}$ can be written in terms of x', y', and z' by subsituting $x' = r_{0,0}x + r_{0,1}y + r_{0,2}z$, and so forth. We can now reproject the rotated version of $\hat{f}$ into the SH basis. Consider, for example, how to compute the new coefficient $\tilde{c}_1^{-1}$:

$$\tilde{c}_1^{-1} = \int_{S^2} \hat{f}(x', y', z')\, Y_1^{-1}(x, y, z)\, \mathrm{d}\omega.$$

If we expand $\hat{f}$ and work through the integral, the result turns out to be a linear combination of original coefficients with entries from the original rotation matrix:

$$\begin{aligned} \tilde{c}_1^{-1} &= r_{1,1}c_1^{-1} + -r_{2,1}c_1^0 + r_{0,1}c_1^1 \\ \tilde{c}_1^0 &= -r_{1,2}c_1^{-1} + -r_{2,2}c_1^0 - r_{0,2}c_1^1 \\ \tilde{c}_1^1 &= r_{1,0}c_1^{-1} + -r_{2,0}c_1^0 + r_{0,0}c_1^1. \end{aligned}$$

This rotation can equivalently be expressed in the following 3×3 SH rotation matrix. (The SH rotation matrix for the $l = 1$ case happens to be 3×3, but recall that higher SH

4 In this section, we will use R to denote 3×3 coordinate system matrices and S to denote SH rotation matrices, which are in general of dimension $(l_{\max} + 1) \times (l_{\max} + 1)$.

bands will have larger rotation matrices.)

$$\begin{pmatrix} \tilde{c}_1^{-1} \\ \tilde{c}_1^{0} \\ \tilde{c}_1^{1} \end{pmatrix} = \begin{pmatrix} r_{11} & -r_{21} & r_{01} \\ -r_{12} & r_{22} & -r_{02} \\ r_{10} & -r_{20} & r_{00} \end{pmatrix} \begin{pmatrix} c_1^{-1} \\ c_1^{0} \\ c_1^{1} \end{pmatrix}.$$

While this general approach works well for low-order SH (up to $l = 2$ or so), higher order bands lead to dense matrices with computationally complex individual entries—much worse than the simple result in the $l = 1$ case. Therefore, we will instead use an approach that builds arbitrary rotations through a series of individual rotations around coordinate axes.

SH Rotations via Euler Decomposition

All 3×3 rotation matrices can be equivalently represented as three rotations about coordinate axes (where any pair of successive rotations is about different axes). An efficient approach to spherical harmonic rotation can be developed by decomposing arbitrary rotation matrices into three Euler angles, representing rotations about the z, y, and then again z axis. Specifically, given a rotation matrix $\mathbf{R}$, we can write it in terms of rotations about angles α, β, and γ such that

$$\mathbf{R} = \mathbf{R}_z(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_z(\gamma). \quad [17.14]$$

The rotation about the y axis, R_y, can further be expressed as a rotation about the x axis by -90 degrees, a rotation about the z axis, and a rotation about the x axis by 90 degrees:

$$\mathbf{R}_y(\beta) = \mathbf{R}_x(-90)\mathbf{R}_z(\beta)\mathbf{R}_x(90).$$

Expressing R_y in this manner makes it possible for the implementation below to only need to handle arbitrary rotations about z. The final rotation is thus given by

$$\mathbf{R} = \mathbf{R}_z(\alpha)\mathbf{R}_x(-90)\mathbf{R}_z(\beta)\mathbf{R}_x(90)\mathbf{R}_z(\gamma).$$

The `SHRotate()` function takes a 3×3 rotation matrix; performs an Euler angle decomposition to find the angles α, β, and γ; and then performs the corresponding set of rotations to the given SH coefficients.

⟨*Spherical Harmonics Definitions*⟩ +≡

```
void SHRotate(const Spectrum *c_in, Spectrum *c_out, const Matrix4x4 &m,
              int lmax, MemoryArena &arena) {
    float alpha, beta, gamma;
    toZYZ(m, &alpha, &beta, &gamma);
    Spectrum *work = arena.Alloc<Spectrum>(SHTerms(lmax));
    SHRotateZ(c_in, c_out, gamma, lmax);
    SHRotateXPlus(c_out, work, lmax);
    SHRotateZ(work, c_out, beta, lmax);
    SHRotateXMinus(c_out, work, lmax);
    SHRotateZ(work, c_out, alpha, lmax);
}
```

Matrix4x4 1021
MemoryArena 1015
MemoryArena::Alloc() 1016
SHRotateXMinus() 956
SHRotateXPlus() 956
SHRotateZ() 955
SHTerms() 935
Spectrum 263
toZYZ() 953

We won't include `toZYZ()` here; it is based on Shoemake's implementation (1994b). The basic idea is that the angles for the Euler decomposition of a given matrix can be found using the values of the elements of the matrix. If we expand out the product of the three

rotation matrices from Equation (17.14), we can find expressions for each of the entries of $\mathbf{R}$. For example, $r_{0,2} = \cos\alpha \sin\beta$, $r_{2,2} = \cos\beta$, and so forth. These expressions can in turn be used to solve for the angles.

In order to be able to implement `SHRotateZ()`, we next need to find the matrices that rotate SH coefficients a given angle about the z axis. Thanks to the particular form of the SH functions, rotating a function represented in the spherical harmonic basis about the z axis is particularly straightforward. Given a particular rotation angle α about z, we can find the rotated function by taking the original coefficients for the function $\hat{f}$ and rotating the basis functions by α, using the (θ, ϕ) form of the SH functions. The rotated function is:

$$\hat{f}(\theta, \phi + \alpha) = \sum_{l}^{l_{\max}} \sum_{m=-l}^{l} c_l^m Y_l^m(\theta, \phi + \alpha).$$

Given the rotated function, we can project it back into the SH basis, integrating with the basis functions to find the SH coefficients of the rotated function,

$$\tilde{c}_l^m = \sum_{l}^{l_{\max}} \sum_{m'=-l}^{l} \int_0^{2\pi} \int_0^{\pi} (c_l^{m'} Y_l^{m'}(\theta, \phi + \alpha))\, Y_l^m(\theta, \phi)\, \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi.$$

We can ignore all bands $l' \neq l$ when computing the coefficients $\tilde{c}_l^m$ since the coefficients from any l band don't affect other bands' coefficients under rotation, as mentioned earlier. We can also pull the coefficients $c_l^{m'}$ out of the integrand, giving:

$$\tilde{c}_l^m = \sum_{m'=-l}^{l} c_l^{m'} \int_0^{2\pi} \int_0^{\pi} Y_l^{m'}(\theta, \phi + \alpha)\, Y_l^m(\theta, \phi)\, \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi.$$

The values of these integrals can be found ahead of time, allowing us to express the rotation as a matrix multiplication.

For the z axis rotation case, these integrals of the products $Y_l^m(\theta, \phi) Y_l^{m'}(\theta, \phi + \alpha)$ have a particularly simple form. Again, for $l = 1$, working out the integrals gives

$$\begin{pmatrix} \tilde{c}_1^{-1} \\ \tilde{c}_1^0 \\ \tilde{c}_1^1 \end{pmatrix} = \begin{pmatrix} \cos -\alpha & 0 & \sin -\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{pmatrix} \begin{pmatrix} c_1^{-1} \\ c_1^0 \\ c_1^1 \end{pmatrix}.$$

As l increases, the matrix takes on a regular form. For $l = 2$, it is:

$$\begin{pmatrix} \cos -2\alpha & 0 & 0 & 0 & \sin -2\alpha \\ 0 & \cos -\alpha & 0 & \sin -\alpha & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & \sin\alpha & 0 & \cos\alpha & 0 \\ \sin 2\alpha & 0 & 0 & 0 & \cos 2\alpha \end{pmatrix},$$

SHRotateZ() 955

and so forth. For arbitrary l, the rotation matrix elements $s_{i,j}$ are:

$$s_{i,j} = \begin{cases} \cos((j-l)\alpha) & i = j, i \neq l, j \neq l \\ \sin((l-j)\alpha) & i + j = 2l, i \neq l, j \neq l \\ 1 & i = j = l \\ 0 & \text{otherwise.} \end{cases} \quad [17.15]$$

The SHRotateZ() implements this rotation for Spectrum-valued spherical harmonic coefficients. It rotates all of the bands by the angle α, up to the given $l_{\max}$ band, storing the results in the memory pointed to by c_out. Note that the first coefficient, representing the constant $l = 0$ term, is unchanged by rotation.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHRotateZ(const Spectrum *c_in, Spectrum *c_out, float alpha,
               int lmax) {
    c_out[0] = c_in[0];
    if (lmax == 0) return;
    ⟨Precompute sine and cosine terms for z-axis SH rotation 955⟩
    for (int l = 1; l <= lmax; ++l) {
        ⟨Rotate coefficients for band l about z 955⟩
    }
}
```

This function first computes all of the sine and cosine values needed for all of the SH bands up to $l_{\max}$ using the efficient sinCosIndexed() function.

⟨*Precompute sine and cosine terms for z-axis SH rotation*⟩ ≡ 955
```
float *ct = ALLOCA(float, lmax+1);
float *st = ALLOCA(float, lmax+1);
sinCosIndexed(sinf(alpha), cosf(alpha), lmax+1, st, ct);
```

The rotations are then performed by directly implementing the matrix/vector multiplication from the matrices given by Equation (17.15). Generating the complete rotation matrix and using a general matrix/vector multiplication routine would be substantially less efficient, given the sparsity of the full rotation matrices.

⟨*Rotate coefficients for band* l *about z*⟩ ≡ 955
```
for (int m = -l; m < 0; ++m)
    c_out[SHIndex(l, m)] =
        ( ct[-m] * c_in[SHIndex(l,  m)] +
         -st[-m] * c_in[SHIndex(l, -m)]);
c_out[SHIndex(l, 0)] = c_in[SHIndex(l, 0)];
for (int m = 1; m <= l; ++m)
    c_out[SHIndex(l, m)] =
        (ct[m] * c_in[SHIndex(l,  m)] +
         st[m] * c_in[SHIndex(l, -m)]);
```

ALLOCA() 1009
SHIndex() 935
sinCosIndexed() 940
Spectrum 263

The last step is to find the x axis rotations by ± 90 degrees in order to implement SHRotateXPlus() and SHRotateXMinus(). These also turn out to be relatively simple matrices. For example, following the same approach working through the integrals of products of SH basis functions, the SH rotation matrix for rotating by 90 degrees about

the x axis in the $l = 2$ case is given by

$$\begin{pmatrix} \tilde{c}_2^{-2} \\ \tilde{c}_2^{-1} \\ \tilde{c}_2^{0} \\ \tilde{c}_2^{1} \\ \tilde{c}_2^{2} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1/2 & 0 & -\sqrt{3}/2 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{3}/2 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} c_2^{-2} \\ c_2^{-1} \\ c_2^{0} \\ c_2^{1} \\ c_2^{2} \end{pmatrix}.$$

Similar to the z rotation case, rotating a given set of coefficients with this matrix can be implemented extremely efficiently because many of the entries are zero, one, or negative one. The terms with square roots can be computed ahead of time, leading to an implementation requiring just a few multiplies and adds.

The rotation matrices for other l bands as well as the rotation matrices for -90 degree rotations around the x axis are similarly sparse; we won't include them here. The routines `SHRotateXPlus()` and `SHRotateXMinus()` implement these rotations.

⟨*Spherical Harmonics Declarations*⟩ +≡

```
void SHRotateXMinus(const Spectrum *c_in, Spectrum *c_out, int lmax);
void SHRotateXPlus(const Spectrum *c_in, Spectrum *c_out, int lmax);
```

17.3 RADIANCE PROBES

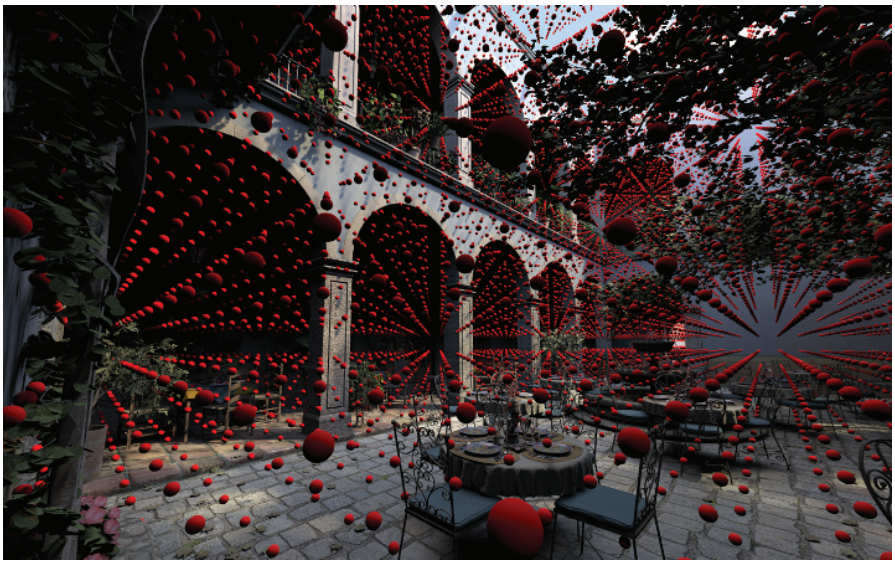
It's often possible to partition the geometry of complex environments into two subsets: static geometry with fixed position and material properties and dynamic geometry with arbitrary positions, materials, and shapes. (Consider, for example, a game level with fixed buildings and vegetation and characters moving through it.) For these sorts of scenes, one approach for interactive rendering is to precompute a representation of directionally varying incident radiance at a set of points in the scene, taking into account light interreflection with the static geometry. When the scene is later rendered, these sampled precomputed radiance functions are used to approximate incident illumination. In this section, we will refer to the technique as *radiance probes*, though it's sometimes also described as *volumetric lighting* or *irradiance volumes*. Figure 17.7 shows a scene rendered with this technique.

There are a few obvious sources of error from this approach. First, incident radiance computed at one point may not be an accurate approximation to incident radiance at other nearby points. (Consider, for example, the difference between the incident radiance function near a wall versus the incident radiance function at a point inside the wall.) Second, the effect of the dynamic objects in the scene isn't represented in the radiance functions used for shading; for example, interreflection between the static objects and the dynamic objects is not accounted for. In spite of these shortcomings, this approach has seen wide use for interactive rendering, as it allows the incorporation of global lighting effects with relatively little computational cost.

17.3.1 CREATING RADIANCE PROBES

Spectrum 263

One advantage of using a physically based renderer like `pbrt` for creating radiance probes is that global illumination algorithms can be used to compute the distribu-



(a)



(b)

Figure 17.7: Rendering with Radiance Probes. (a) San Miguel scene (rendered with a standard direct lighting calculation), showing points at which radiance probes were computed with red spheres. (b) Scene rendered using the radiance probes to approximate the incident illumination. Global lighting effects like indirect lighting from the walls are captured by the radiance probes and low-frequency shadows like those under the tables are visible. However, more detailed illumination effects can not be accurately represented with this sampling density. *(Model courtesy of Guillermo M. Leal Llaguno.)*

tion of illumination in the scene. We will first describe the implementation of the CreateRadianceProbes Renderer, which computes incident radiance and projects it into the spherical harmonics at a grid of points in the scene. It is defined in the files renderers/createprobes.h and renderers/createprobes.cpp. The computed SH coefficients include the effect of *all* of the lights in the scene; thus, there is no incremental performance cost for a large number of lights at final rendering time. Of course, a limited number of SH coefficients can represent only a limited amount of lighting complexity. However, for low-frequency (i.e., diffuse or only slightly glossy) BRDFs, the error from this approximation is often acceptable.

This renderer then writes the SH coefficients out to a file. CreateRadianceProbes is thus a renderer that doesn't create an image as its output but instead computes a series of measurements of the scene.

Section 17.3.2 demonstrates one method for using radiance probes for rendering with the UseRadianceProbes integrator. In general, the greatest benefit from radiance probes comes from their use in an interactive renderer rather than an offline renderer like pbrt; however, here we show their use in pbrt in the interests of simplicity.

The parameters to the CreateRadianceProbes constructor manage the radiance probe creation process. Radiance probes are created with no more than probeSpacing distance between samples in each dimension, in a regular grid with extent given by the provided bounding box. (If an explicit bounding box isn't provided by the parameters in the scene description file, then the scene bounds are used for the bounding box.) The incident radiance functions are projected into spherical harmonics using lmax SH bands and stored in the given file. The camera and integrators from the scene description file are passed to the constructor, and the number of Monte Carlo samples to use when estimating SH coefficients of indirect illumination is given by nIndirSamples.

⟨*CreateRadianceProbes Public Methods*⟩ ≡
```
CreateRadianceProbes(SurfaceIntegrator *surf, VolumeIntegrator *vol,
    const Camera *camera, int lmax, float probeSpacing, const BBox &bbox,
    int nIndirSamples, bool includeDirect, bool includeIndirect,
    float time, const string &filename);
```

The implementation of the constructor is straightforward and isn't included here—the member variables here are set directly from the provided parameters.

⟨*CreateRadianceProbes Private Data*⟩ ≡
```
SurfaceIntegrator *surfaceIntegrator;
VolumeIntegrator *volumeIntegrator;
const Camera *camera;
int lmax, nIndirSamples;
BBox bbox;
bool includeDirectInProbes, includeIndirectInProbes;
float time, probeSpacing;
string filename;
```

BBox 70
Camera 302
CreateRadianceProbes 958
Renderer 24
SurfaceIntegrator 740
UseRadianceProbes 965
VolumeIntegrator 876

As an implementation of the Renderer abstract base class, CreateRadianceProbes must implement the Li() and Transmittance() methods. Their implementations, not in-

cluded here, are essentially the same as the corresponding SamplerRenderer methods defined in Section 1.3.4, with the surfaceIntegrator and volumeIntegrator used to compute the returned values. These methods of CreateRadianceProbes will be called by the SHProjectIncidentIndirectRadiance() function, which is used computing the SH coefficients for the probes below.

After some general preparation, the bulk of the work done by the Render() method is farmed out to tasks that run in parallel; one task is launched for each of the points at which a radiance probe is to be computed. After the tasks have all finished, the implementation here writes the SH coefficients of the probes to a file.

⟨*CreateRadianceProbes Method Definitions*⟩ ≡
```
void CreateRadianceProbes::Render(const Scene *scene) {
    ⟨Compute scene bounds and initialize probe integrators 959⟩
    ⟨Compute sampling rate in each dimension 959⟩
    ⟨Allocate SH coefficient vector pointers for sample points 960⟩
    ⟨Compute random points on surfaces of scene 962⟩
    ⟨Launch tasks to compute radiance probes at sample points⟩
    ⟨Write radiance probe coefficients to file⟩
}
```

If a bounding box for the radiance probes isn't provided in the scene description file, the BBox is initialized to a degenerate bounding box, which has the respective minimum position components greater than the maximum components. In that case, the bounding box for the probes is initialized here from the scene bounding box. Next, the integrators have a chance to do precomputation now that the Scene is available.

⟨*Compute scene bounds and initialize probe integrators*⟩ ≡ 959
```
if (bbox.pMin.x > bbox.pMax.x)
    bbox = scene->WorldBound();
surfaceIntegrator->Preprocess(scene, camera, this);
volumeIntegrator->Preprocess(scene, camera, this);
Sample *origSample = new Sample(NULL, surfaceIntegrator, volumeIntegrator,
                                scene);
```

Now that the bounding box is known, it is possible to determine how many probes to compute. The number of probes in each dimension is set so that the probes are no farther apart than the given probeSpacing.

⟨*Compute sampling rate in each dimension*⟩ ≡ 959
```
Vector delta = bbox.pMax - bbox.pMin;
int nProbes[3];
for (int i = 0; i < 3; ++i)
    nProbes[i] = max(1, Ceil2Int(delta[i] / probeSpacing));
```

Given the number of probes, it's now possible to allocate space for the SH coefficients. The tasks will initialize these values for each probe.

BBox 70
BBox::pMax 71
BBox::pMin 71
Ceil2Int() 1002
CreateRadianceProbes::probeSpacing 958
CreateRadianceProbes::surfaceIntegrator 958
CreateRadianceProbes::volumeIntegrator 958
Integrator::Preprocess() 740
Sample 343
SamplerRenderer 25
Scene 22
Scene::WorldBound() 24
SHProjectIncidentIndirectRadiance() 950
Vector 57



(a) (b)

Figure 17.8: Artifacts From Bad Placement of Radiance Probes. (a) If the incident radiance functions for radiance probes are computed at the centers of their respective grid cells, then some probes will often end up being completely inside closed objects and will be completely black. (b) When these probes are used for rendering, characteristic dark splotches appear when those probes are used to shade points outside the closed objects. When the alternative probe placement approach implemented in this section is used, we get the better result shown in Figure 17.7(b). *(Model courtesy of Guillermo M. Leal Llaguno.)*

⟨*Allocate SH coefficient vector pointers for sample points*⟩ ≡ 959

```
int count = nProbes[0] * nProbes[1] * nProbes[2];
Spectrum **c_in = new Spectrum *[count];
for (int i = 0; i < count; ++i)
    c_in[i] = new Spectrum[SHTerms(lmax)];
```

The most straightforward approach for computing each radiance probe would be to compute the incident radiance function at a single point at the center of the probe's grid cell, as shown in Figure 17.8(a). However, as that figure illustrates, this approach often suffers from sampling errors, since some probes would end up located inside solid objects in the scene. The incident radiance functions for these points would be completely black, and artifacts would occur during rendering when the radiance functions at these probes were used for shading objects that are actually outside the solid objects. Figure 17.8(b) shows these artifacts with the scene from Figure 17.7(b) rendered with the probes placed at the center of each grid cell.

To avoid this issue, `pbrt` implements an approach inspired by Kontkanen and Laine (2006), who investigated this problem but came up with a somewhat different solution than the one implemented here. For each radiance probe to be computed, we first compute the extent of the grid cell of the overall scene bounding box that it represents. We then select a sequence of points inside the cell; for each one, we determine whether it is *indirectly visible* from the camera position. (This is the only requirement for the camera position for this renderer; we just require that it be in the same part of the scene where

Spectrum 263

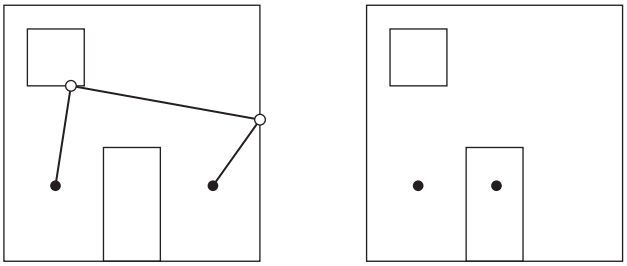


Figure 17.9: Indirect Visibility Between Two Points. (a) Two points (filled circles) are defined to be indirectly visible to each other if there is a path of one or more unobstructed ray segments between them. Here, the vertices of the path on other surfaces of the scene are shown with open circles. (b) These two points aren't indirectly visible to each other since there is no path between them that doesn't pass through the walls of the box that encloses the one on the right.

final rendering will be done.) We define two points as being indirectly visible if there is a path with zero or more vertices on scene surfaces between them; see Figure 17.9. The points in grid cells that are indirectly visible to the camera are assumed to thus not be inside scene objects and can be used to compute the incident radiance function.

This approach doesn't eliminate all sampling errors—see, for example, the case illustrated in the upper right corner of Figure 17.8(a) where a closed piece of geometry splits a cell in half. In this case, the incident radiance functions on the two sides of it are likely be very different. A single radiance probe can't represent this case accurately; using adaptive refinement rather than the regular grid here can reduce the effect of this problem.

In order to compute indirect visibility from the camera position while computing probes, a short precomputation is performed before the tasks are launched. This computation finds a number of points on surfaces in the scene by tracing random paths starting from the camera position; these points are stored in the `surfacePoints` array. If any one of these points has an unobstructed path between it and a candidate radiance probe point, then we know by construction that there is an indirect visibility path from the camera to the point and that the point is a safe one at which to compute a radiance probe (Figure 17.10). Note that this method for computing indirect visibility isn't perfect: there may be some points that are in fact indirectly visible but that are not found by this sampling algorithm. However, we have found this approach to work reasonably well in practice.

The implementation of the fragment ⟨*Generate random path from camera and deposit surface points*⟩ is very similar to the fragment ⟨*Follow ray through scene and attempt to deposit candidate sample points*⟩ defined in Section 16.5.1 (though it doesn't check to make sure that surfaces have BSSRDFs before storing points and doesn't try to enforce a Poisson sphere distribution of points). Therefore, this fragment is not included here.

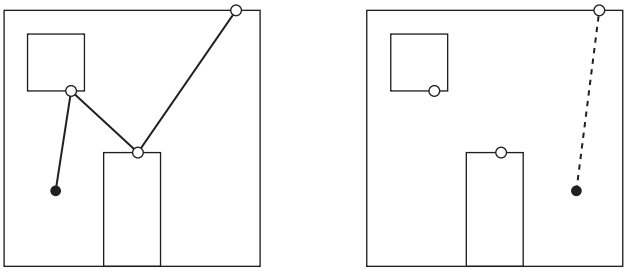


Figure 17.10: Efficiently Computing Indirect Visibility. (a) In a preprocessing step, random paths are followed from the camera (filled circle), and intersection points on scene surfaces (open circles) are recorded in an array. (b) Given a candidate probe location (filled circle), we can efficiently test to see if it is indirectly visible to the camera by tracing shadow rays between it and all of the stored intersection points. If any of the rays is not occluded, the point and the camera are indirectly visible.

⟨*Compute random points on surfaces of scene*⟩ ≡ 959

```
⟨Create scene bounding sphere to catch rays that leave the scene 891⟩
vector<Point> surfacePoints;
uint32_t nPoints = 32768, maxDepth = 32;
surfacePoints.reserve(nPoints + maxDepth);
Point pCamera = camera->CameraToWorld(camera->shutterOpen,
                                      Point(0, 0, 0));
surfacePoints.push_back(pCamera);
RNG rng;
while (surfacePoints.size() < nPoints) {
    ⟨Generate random path from camera and deposit surface points⟩
}
```

After these points are found, tasks are launched to compute the radiance probes. One task is launched for each probe by the fragment ⟨*Launch tasks to compute radiance probes at sample points*⟩, which we won't include in the text here. After the tasks complete, the fragment ⟨*Write radiance probe coefficients to file*⟩ (also not included here) writes a text file that stores the overall bounding box, the number of SH bands, and the SH coefficients for each probe in turn. This file can be read by the `UseRadianceProbes Integrator`, defined shortly.

The task for each probe first computes the bounding box of the subregion of the scene for which it is responsible, initializes a few common variables, and then selects a number of candidate points within the bounding box of its region. For all of the points that are indirectly visible from the given camera position, it samples the incident radiance function at the point and projects it into SH coefficients. In the end, it returns the average SH coefficients for the radiance function for all accepted candidate points. Because the SH functions used here are a linear basis, averaging their coefficients for the probe points used gives the average of the projected incident radiance functions at the points. No more

Camera::CameraToWorld 302
CreateRadianceProbes::camera 958
Integrator 740
Point 63
RNG 1003
UseRadianceProbes 965

than 256 points within the cell are tested, but once 32 points that are indirectly visible from the camera are found the task exits.

```
⟨CreateRadianceProbes Method Definitions⟩ +≡
  void CreateRadProbeTask::Run() {
      ⟨Compute region in which to compute incident radiance probes 963⟩
      ⟨Initialize common variables for CreateRadProbeTask::Run() 963⟩
      for (int i = 0; i < 256; ++i) {
          if (nFound == 32) break;
          ⟨Try to compute radiance probe contribution at ith sample point 963⟩
      }
      ⟨Compute final average value for probe and cleanup 965⟩
  }
```

The bounding box for the cell that corresponds to this probe is found by first computing the integer (sx, sy, sz) coordinates for the cell with respect to the overall probe sampling rate and then linearly interpolating from the overall bounding box corners.

```
⟨Compute region in which to compute incident radiance probes⟩ ≡ 963
  int sx = pointNum % nProbes[0];
  int sy = (pointNum / nProbes[0]) % nProbes[1];
  int sz = pointNum / (nProbes[0] * nProbes[1]);
  float tx0 = float(sx) / nProbes[0], tx1 = float(sx+1) / nProbes[0];
  float ty0 = float(sy) / nProbes[1], ty1 = float(sy+1) / nProbes[1];
  float tz0 = float(sz) / nProbes[2], tz1 = float(sz+1) / nProbes[2];
  BBox b(bbox.Lerp(tx0, ty0, tz0), bbox.Lerp(tx1, ty1, tz1));
```

A number of variables will be needed in the following code. First is a random number generator, uniquely seeded based on the radiance probe we're generating. The c_probe variable stores the SH coefficients computed at a single point before they're added to the average coefficients for all of the probes in this cell, and the MemoryArena is used by the incident radiance projection routines. nFound tracks the number of found probe points that are indirectly visible to the camera, and lastVisibleOffset, described further below, is used to accelerate the indirect visibility tests.

```
⟨Initialize common variables for CreateRadProbeTask::Run()⟩ ≡ 963
  RNG rng(pointNum);
  Spectrum *c_probe = new Spectrum[SHTerms(lmax)];
  MemoryArena arena;
  uint32_t nFound = 0, lastVisibleOffset = 0;
```

After computing the position of each candidate sample point within the probe's bounds, the code checks to see if the point is indirectly visible from the given camera point; if so, it computes the SH coefficients for the incident radiance function at the point.

```
⟨Try to compute radiance probe contribution at ith sample point⟩ ≡ 963
  ⟨Compute ith candidate point p in cell's bounding box 964⟩
  ⟨Skip point p if not indirectly visible from camera 964⟩
  ⟨Compute SH coefficients of incident radiance at point p 964⟩
```

BBox 70
BBox::Lerp() 73
MemoryArena 1015
RNG 1003
SHTerms() 935
Spectrum 263

To choose candidate points, points from a 3D Halton sequence (Section 7.4.2) are used to interpolate between the corners of the cell's bounding box.

⟨*Compute* `i` *th candidate point* `p` *in cell's bounding box*⟩ ≡ 963
```
float dx = RadicalInverse(i+1, 2);
float dy = RadicalInverse(i+1, 3);
float dz = RadicalInverse(i+1, 5);
Point p = b.Lerp(dx, dy, dz);
```

For each candidate point `p`, we need to see if it's indirectly visible from the camera: if any of the points in the `surfacePoints` array and the candidate point are mutually visible, then the candidate point can be used. The implementation here keeps track of the offset to the last point from `surfacePoints` that had an unoccluded path to a previous candidate point in this probe's grid cell in `lastVisibleOffset`. It then tests visibility between that point and the next candidate point first. It's often the case that this point will also have an unoccluded path to the next candidate point, so testing it first can substantially speed up this part of the computation.

⟨*Skip point* `p` *if not indirectly visible from camera*⟩ ≡ 963
```
if (scene->IntersectP(Ray(surfacePoints[lastVisibleOffset],
                          p - surfacePoints[lastVisibleOffset],
                          1e-4f, 1.f, time))) {
    uint32_t j;
    ⟨See if point is visible to any element of surfacePoints 964⟩
    if (j == surfacePoints.size())
        continue;
}
++nFound;
```

⟨*See if point is visible to any element of* `surfacePoints`⟩ ≡ 964
```
for (j = 0; j < surfacePoints.size(); ++j)
    if (!scene->IntersectP(Ray(surfacePoints[j], p - surfacePoints[j],
                               1e-4f, 1.f, time))) {
        lastVisibleOffset = j;
        break;
    }
```

Given a valid point inside the probe's grid cell, the utility functions defined in Section 17.2.3 are used to compute SH coefficients for the incident radiance function at the point, accumulating the results into the `c_in` array.

⟨*Compute SH coefficients of incident radiance at point* `p`⟩ ≡ 963
```
if (includeDirectInProbes) {
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_probe[i] = 0.f;
    SHProjectIncidentDirectRadiance(p, 0.f, time, arena, scene,
                                    true, lmax, rng, c_probe);
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_in[i] += c_probe[i];
}
```

BBox::Lerp() 73
MemoryArena::FreeAll() 1017
Point 63
RadicalInverse() 362
Ray 66
Scene::IntersectP() 24
SHProjectIncidentDirectRadiance() 949
SHProjectIncidentIndirectRadiance() 950
SHTerms() 935

```
if (includeIndirectInProbes) {
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_probe[i] = 0.f;
    SHProjectIncidentIndirectRadiance(p, 0.f, time, renderer,
        origSample, scene, lmax, rng, nIndirSamples, c_probe);
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_in[i] += c_probe[i];
}
arena.FreeAll();
```

The final coefficient values in `c_in` for this cell are found by dividing the summed coefficient values by the number of points at which probes were computed.

⟨*Compute final average value for probe and cleanup*⟩ ≡ 963

```
if (nFound > 0)
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_in[i] /= nFound;
delete[] c_probe;
```

17.3.2 USING RADIANCE PROBES

The `UseRadianceProbes` `SurfaceIntegrator` reads in a text file of radiance probes generated by the `CreateRadianceProbes` `Renderer` and uses the probes to illuminate the scene. Because the lighting computations with radiance probes can be performed extremely efficiently and because looking up radiance probes stored in a regular grid is straightforward, the corresponding lighting calculations can reasonably be performed in interactive rendering systems.

The `UseRadianceProbes` constructor (not included here) reads in the values of its member variables from the given radiance probe file.

⟨*UseRadianceProbes Private Data*⟩ ≡

```
BBox bbox;
int lmax, includeDirectInProbes, includeIndirectInProbes;
int nProbes[3];
Spectrum *c_in;
```

BBox 70
CreateRadianceProbes 958
Renderer 24
SHTerms() 935
Spectrum 263
SurfaceIntegrator 740
UniformSampleAllLights() 745

The structure of the `UseRadianceProbes::Li()` method is similar to that of most other `SurfaceIntegrators`. Here, we'll only include the fragments that handle illumination from radiance probes. If the probes only include indirect lighting, then the implementation performs the regular direct lighting calculation with `UniformSampleAllLights()`. (Interactive renderers that use radiance probes often use conventional methods like shadow maps to compute direct lighting and only use the radiance probes for indirect illumination. Or, they may include the contribution of some direct lights in the radiance probes and compute the effect of others explicitly.)

⟨*Compute reflection for radiance probes integrator*⟩ ≡
```
if (!includeDirectInProbes)
    L += UniformSampleAllLights(scene, renderer, arena, p, n,
            wo, isect.rayEpsilon, ray.time, bsdf, sample, rng,
            lightSampleOffsets, bsdfSampleOffsets);
```
⟨*Compute reflected lighting using radiance probes* 966⟩

To use the radiance probes for lighting calculations, the first step is to look up the probes around the point being shaded and to trilinearly interpolate their coefficients to get coefficients that approximate the incident radiance function at the lookup point. In this section, we'll implement an efficient method that computes the incident irradiance from this interpolated incident radiance function and then uses it to shade the surface, approximating its full BSDF with a diffuse term.

⟨*Compute reflected lighting using radiance probes*⟩ ≡ 966
⟨*Compute probe coordinates and offsets for lookup point* 966⟩
⟨*Get radiance probe coefficients around lookup point* 966⟩
⟨*Compute incident radiance from radiance probe coefficients* 967⟩
⟨*Convolve incident radiance to compute irradiance function* 968⟩
⟨*Evaluate irradiance function and accumulate reflection* 969⟩

The integer coordinates of the lookup point `p` are found using the same calculation as is used to convert from continuous to discrete pixel coordinates (recall the discussion in Section 7.1.7). Given the discrete coordinates, floating-point offsets `dx`, `dy`, and `dz` are found for trilinear interpolation.

⟨*Compute probe coordinates and offsets for lookup point*⟩ ≡ 966
```
Vector offset = bbox.Offset(p);
float voxx = (offset.x * nProbes[0]) - 0.5f;
float voxy = (offset.y * nProbes[1]) - 0.5f;
float voxz = (offset.z * nProbes[2]) - 0.5f;
int vx = Floor2Int(voxx), vy = Floor2Int(voxy), vz = Floor2Int(voxz);
float dx = voxx - vx, dy = voxy - vy, dz = voxz - vz;
```

The `c_inXYZ()` utility function, not included here in the book text, clamps the given discrete coordinates to the valid range and returns a pointer to the first `Spectrum` in the array that holds the SH coefficients at the given grid cell.

⟨*Get radiance probe coefficients around lookup point*⟩ ≡ 966
```
const Spectrum *b000 = c_inXYZ(lmax, vx,   vy,   vz);
const Spectrum *b100 = c_inXYZ(lmax, vx+1, vy,   vz);
const Spectrum *b010 = c_inXYZ(lmax, vx,   vy+1, vz);
const Spectrum *b110 = c_inXYZ(lmax, vx+1, vy+1, vz);
const Spectrum *b001 = c_inXYZ(lmax, vx,   vy,   vz+1);
const Spectrum *b101 = c_inXYZ(lmax, vx+1, vy,   vz+1);
const Spectrum *b011 = c_inXYZ(lmax, vx,   vy+1, vz+1);
const Spectrum *b111 = c_inXYZ(lmax, vx+1, vy+1, vz+1);
```

BBox::Offset() 73
Floor2Int() 1002
Intersection::rayEpsilon 186
Ray::time 67
Spectrum 263
UniformSampleAllLights() 745
UseRadianceProbes::includeDirectInProbes 965
UseRadianceProbes::lmax 965
UseRadianceProbes::nProbes 965
Vector 57

Given the pointers to the eight sets of coefficients around the point, we'll now compute the trilinearly interpolated SH coefficients that represent the incident radiance at p and store them in the temporary c_inp array.

⟨*Compute incident radiance from radiance probe coefficients*⟩ ≡ 966
```
Spectrum *c_inp = arena.Alloc<Spectrum>(SHTerms(lmax));
for (int i = 0; i < SHTerms(lmax); ++i) {
    ⟨Do trilinear interpolation to compute SH coefficients at point 967⟩
}
```

⟨*Do trilinear interpolation to compute SH coefficients at point*⟩ ≡ 967
```
Spectrum c00 = Lerp(dx, b000[i], b100[i]);
Spectrum c10 = Lerp(dx, b010[i], b110[i]);
Spectrum c01 = Lerp(dx, b001[i], b101[i]);
Spectrum c11 = Lerp(dx, b011[i], b111[i]);
Spectrum c0 = Lerp(dy, c00, c10);
Spectrum c1 = Lerp(dy, c01, c11);
c_inp[i] = Lerp(dz, c0, c1);
```

Given the incident radiance function in SH coefficients, we now need to compute the reflected radiance due to illumination at the point being shaded. Here, we will show how to compute the incident irradiance function from the incident radiance function using efficient methods for convolution of functions in the spherical harmonic basis. The incident irradiance is in turn used to compute the reflected radiance under the simplification of treating all surfaces as diffuse. (Recall from Equation (15.13) that the reflected radiance from a diffuse surface is equal to the irradiance arriving at the surface times its diffuse reflectance.) This general approach is frequently used with radiance probes in interactive applications.

Ramamoorthi (2002) showed that the reflection equation, Equation (5.8), can equivalently be written as a convolution of a signal (the incident radiance function) with a filter (the cosine weighted BSDF). If both of these are represented in the SH basis, then the convolution can be performed very efficiently directly in the SH basis, similarly to how convolutions in 2D Cartesian coordinates can be performed very efficiently with the product of Fourier basis coefficients (Equation (7.3)). This approach leads to a number of new efficient algorithms for computing reflection.

Here, we would like to compute the incident irradiance function on one side of a surface, which is given as a function of position and normal by the integral

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i})(\cos\theta_\mathrm{i})\, \mathrm{d}\omega_\mathrm{i} = \int_{\mathcal{S}^2} L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i}) \max(0, \cos\theta_\mathrm{i})\, \mathrm{d}\omega_\mathrm{i}.$$

Ramamoorthi showed that, if the BRDF can be reparameterized to be a 1D function of θ, the SH coefficients of the convolution of the BRDF and the incident radiance function function have a particularly simple form. Given a function of direction $f(\omega)$ and a circularly symmetric 1D function $g(\theta)$, with respective SH coefficients f_l^m and g_l^m, then it's possible to directly compute the SH coefficients of the function that is the result of their convolution $(f * g)$,

$$(f * g)_l^m = \Lambda_l f_l^m g_l^0, \tag{17.16}$$

Lerp() 1000
MemoryArena::Alloc() 1016
SHTerms() 935
Spectrum 263

where

$$\Lambda_l = \sqrt{\frac{4\pi}{2l+1}}.$$

(Note that $g_l^m = 0$ for $m \neq 0$ thanks to the 1D symmetry condition.)

For computing irradiance, no reparameterization is necessary, since the filter function, $g(\theta) = \max(0, \cos\theta)$, is already 1D. The `SHConvolveCosTheta()` function implements the convolution of Equation (17.16) for the $\max(0, \cos\theta)$ function, returning the SH coefficients c_l^m for incident irradiance from the SH coefficients of incident radiance. The resulting irradiance function is a function of the surface normal direction:

$$E(\mathrm{p}, \mathbf{n}) \approx \sum_{l=0}^{l_{\max}} \sum_{m=-l}^{l} c_l^m Y_l^m(\mathbf{n}). \tag{17.17}$$

⟨*Convolve incident radiance to compute irradiance function*⟩ ≡ 966
```
Spectrum *c_E = arena.Alloc<Spectrum>(SHTerms(lmax));
SHConvolveCosTheta(lmax, c_inp, c_E);
```

The SH coefficients g_l^m for the projected $\max(0, \cos\theta)$ function can be computed in closed form. For computational efficiency, we have precomputed them for l up to 17; this is substantially many more terms than are necessary in practice—Ramamoorthi has shown that just the bands up to $l = 2$ can be used to represent irradiance with very low error. All coefficients for $l > 17$ are therefore clamped to zero here.

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHConvolveCosTheta(int lmax, const Spectrum *c_in,
                        Spectrum *c_out) {
    static const float c_costheta[18] = { 0.8862268925, 1.0233267546,
        0.4954159260, 0.0000000000, -0.1107783690, 0.0000000000,
        0.0499271341, 0.0000000000, -0.0285469331, 0.0000000000,
        0.0185080823, 0.0000000000, -0.0129818395, 0.0000000000,
        0.0096125342, 0.0000000000, -0.0074057109, 0.0000000000 };
    for (int l = 0; l <= lmax; ++l)
        for (int m = -l; m <= l; ++m) {
            int o = SHIndex(l, m);
            if (l < 18) c_out[o] = lambda(l) * c_in[o] * c_costheta[l];
            else        c_out[o] = 0.f;
        }
}
```

⟨*Spherical Harmonics Local Definitions*⟩ +≡
```
static inline float lambda(float l) {
    return sqrtf((4.f * M_PI) / (2.f * l + 1.));
}
```

Now that we have the SH coefficients of the irradiance function `c_E`, we need to compute the value of the irradiance function for the normal $\mathbf{n}$ using Equation (17.17). The diffuse reflectance ρ times the reconstructed irradiance gives the final contribution to outgoing radiance.

lambda() 968
MemoryArena::Alloc() 1016
M_PI 1002
SHConvolveCosTheta() 968
SHIndex() 935
SHTerms() 935
Spectrum 263

⟨*Evaluate irradiance function and accumulate reflection*⟩ ≡ 966
```
Spectrum rho = bsdf->rho(wo, rng, BSDF_ALL_REFLECTION);
float *Ylm = ALLOCA(float, SHTerms(lmax));
SHEvaluate(Vector(Faceforward(n, wo)), lmax, Ylm);
Spectrum E = 0.f;
for (int i = 0; i < SHTerms(lmax); ++i)
    E += c_E[i] * Ylm[i];
L += rho * INV_PI * E.Clamp();
```

It's also possible to convolve incident radiance with the (nonphysical) Phong BRDF

$$f_r(p, \omega_o, \omega_i) = \frac{n+1}{2\pi}(\omega' \cdot \omega_i)^n,$$

(where ω' is the direction of the outgoing vector ω_o reflected about the surface normal $\mathbf{n}$) to get a reflected radiance function as a function of the specular reflected direction. Though this functionality isn't currently used in `pbrt`, the `SHConvolvePhong()` function, not included in the text here, implements this computation for $g(\theta) = (\cos\theta)^n$.

17.4 PRECOMPUTED DIFFUSE TRANSFER

Precomputed radiance transfer (PRT) methods generally try to precompute information about how objects reflect and shadow light for use at rendering time, given arbitrary incident lighting distributions. For example, given a static geometric model of a tree, one might want to store a representation of how much light reaches the leaves in the interior of the tree as a function of the illumination arriving from outside of it. Ideally, this representation would encode all shadowing and inter-reflection effects that the light undergoes.

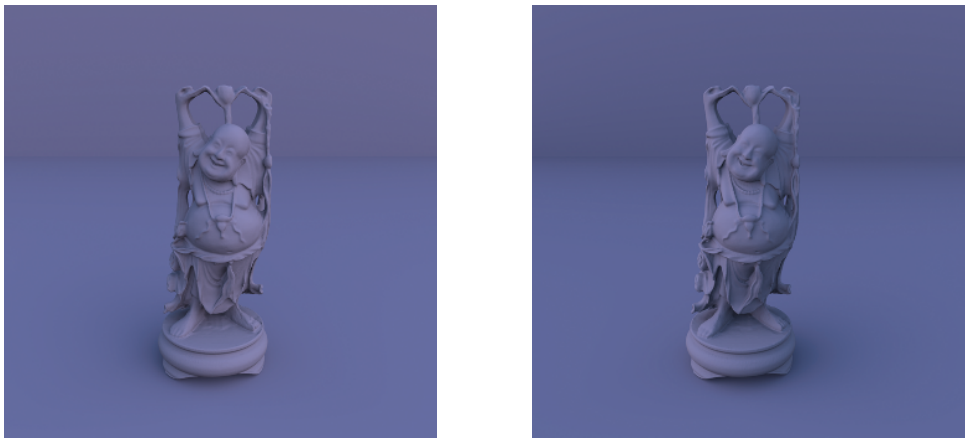
A wide variety of PRT techniques have been developed. Different techniques make different simplifying assumptions and in turn place different limitations on the lighting effects that they can handle. This section and the following section will describe the implementations of two such methods.

In general, PRT techniques assume that some representation of the directionally varying incident radiance function is available for each object in the scene to be shaded. This lighting information is generally represented in some basis; here, we will continue to use the spherical harmonics. The incident lighting information comes from one of a number of sources:

- A single infinite area light source (in which case all objects in the scene have the same incident lighting)
- A collection of radiance probes, such as were implemented in the previous section
- A cubical environment map of a dynamic scene rendered with the center of the object as its origin

Hybrid approaches are possible as well, where a single infinite area light source might be used to illuminate the entire scene, but where a custom incident radiance function is computed for each object to be shaded by modifying the incident radiance function

ALLOCA() 1009
BSDF::rho() 482
BSDF_ALL_REFLECTION 428
Faceforward() 66
INV_PI 1002
SHConvolvePhong() 969
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Spectrum::Clamp() 265
Vector 57



(a) (b)

Figure 17.11: Buddha Model Rendered Using Diffuse Precomputed Radiance Transfer. A view of the Buddha model illuminated by an infinite area light source, where the light source has been rotated between the two frames. Rendering was performed with the `DiffusePRTIntegrator`, which implements algorithms that make it possible to re-render images like these extremely efficiently after the initial precomputation has been performed.

to account for nearby objects that block illumination from the infinite light source. The "Further Reading" section has more details on these approaches.

The `DiffusePRTIntegrator` implements one of the simplest PRT methods, described by Sloan et al. (2002). It is limited to diffuse surfaces and is based on encoding how a geometric model diffusely reflects arbitrary incident illumination, accounting for self-shadowing (and optionally interreflection). This approach is extremely efficient, requiring only a small amount of computation to evaluate the approximation to the scattering equation during rendering.

Figure 17.11 shows two images rendered with the `DiffusePRTIntegrator`. While these images are not any more complex than those rendered with offline ray tracing by other integrators in `pbrt`, remember that once the offline precomputation has been performed images like these can easily be rendered interactively at hundreds of times per second, with both the illumination and viewing position varying from frame to frame.

Diffuse PRT is based on a few simplifications to the scattering equation, Equation (5.8). If we ignore for now multiply reflected light, we have

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) \, |\cos \theta_i| \, d\omega_i.$$

With diffuse PRT, diffuse reflection is assumed, so that scattered radiance isn't directionally varying and the BSDF, ρ/π, is constant and can be moved outside the integral:

`DiffusePRTIntegrator` 972

$$L_o(p) = \frac{\rho(p)}{\pi} \int_{S^2} L_i(p, \omega_i) \, |\cos \theta_i| \, d\omega_i.$$

If we assume an infinitely distant environment light source, then the integrand can be described by the product of a directionally varying incident radiance term $L_{\text{env}}(\omega)$ that ignores occlusion between the point and the light and a transfer function T that represents visiblility and the cosine factor, giving

$$L_{\text{o}}(\text{p}) = \frac{\rho(\text{p})}{\pi} \int_{\mathbb{S}^2} L_{\text{env}}(\omega_{\text{i}})\, T(\text{p}, \omega_{\text{i}})\, \text{d}\omega_{\text{i}}, \tag{17.18}$$

where the *transfer function* T is defined by

$$T(\text{p}, \omega_{\text{i}}) = V(\text{p}, \omega_{\text{i}})\, |\cos \theta_{\text{i}}|. \tag{17.19}$$

Here, $V(\text{p}, \omega_{\text{i}})$ is a visibility term that is one if the ray from p in direction ω is unoccluded and zero otherwise. The transfer function essentially describes how incident radiance from the light is mapped to incident radiance at a point.[5]

Note that the transfer function depends purely on the geometry of the model being rendered and can thus be precomputed independently of the actual lighting environment. Furthermore, the $L_{\text{env}}(\omega)$ term depends purely on the incident lighting environment at rendering time. Sloan et al. observed that if both the incident radiance function and the transfer function are represented in SH coefficients (L_l^m and T_l^m, respectively), then the integral in Equation (17.18) can be evaluated simply by summing the products of their respective basis function coefficients (recall Equation (17.4)):

$$L_{\text{o}}(\text{p}) \approx \frac{\rho(\text{p})}{\pi} \sum_{l=0}^{l_{\max}} \sum_{m=-l}^{l} L_l^m T_l^m. \tag{17.20}$$

To apply this approach to interactive rendering generally requires that the transfer function's SH coefficients be precomputed in an offline process. They are generally computed at the vertices of polygonal models, or at the texels of a texture mapped to the model. At runtime, the transfer function's SH coefficients are looked up, the incident radiance function's SH coefficients are computed (generally only once per model or frame), and the efficient computation of Equation (17.20) is applied. In the implementation here in the `DiffusePRTIntegrator`, the transfer function coefficients are found at each point immediately before it is shaded with Equation (17.20). Thus, the efficiency benefits aren't experienced when using this integrator, but this simplifies the presentation here.

The `DiffusePRTIntegrator`'s constructor takes the number of SH bands to use in its lighting calculations and a sample count used in the Monte Carlo routines that compute SH coefficients. A single set of SH coefficients is used for the incident radiance function for all objects in the implementation here; they are initialized in the `Preprocess()` method below.

`DiffusePRTIntegrator` 972

5 The transfer function can be generalized to account for light that is reflected by other parts of the model before arriving at the current point; this extension is left as an exercise at the end of the chapter.

⟨*DiffusePRTIntegrator Method Definitions*⟩ ≡
```
DiffusePRTIntegrator::DiffusePRTIntegrator(int lm, int ns)
    : lmax(lm), nSamples(RoundUpPow2(ns)) {
    c_in = new Spectrum[SHTerms(lmax)];
}
```

⟨*DiffusePRTIntegrator Private Data*⟩ ≡
```
const int lmax, nSamples;
Spectrum *c_in;
```

The SH coefficients for incident radiance from the lights in the scene are computed at a single point at the center of the scene's bounding box, ignoring occlusion between that point and the light sources. (Recall that the transfer function includes a visibility factor.)

⟨*DiffusePRTIntegrator Method Definitions*⟩ +≡
```
void DiffusePRTIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    BBox bbox = scene->WorldBound();
    Point p = .5f * bbox.pMin + .5f * bbox.pMax;
    RNG rng;
    MemoryArena arena;
    SHProjectIncidentDirectRadiance(p, 0.f, camera->shutterOpen, arena,
                                    scene, false, lmax, rng, c_in);
}
```

The `Li()` method starts by declaring the usual set of common variables and computing the BSDF. The heart of its work is performed by the ⟨*Compute reflected radiance using diffuse PRT*⟩ fragment, which both computes the information needed for diffuse PRT at a point in the scene and then uses this information for the lighting calculation.

⟨*Compute reflected radiance using diffuse PRT*⟩ ≡
⟨*Project diffuse transfer function at point to SH* 972⟩
⟨*Compute integral of product of incident radiance and transfer function* 973⟩

This computation of the SH coefficients of the transfer function is encapsulated in a utility function under the expectation that other implementations may want to use this to precompute values at vertices of the model or at other sampling rates. Here, the coefficients are computed at the ray's intersection point.

⟨*Project diffuse transfer function at point to SH*⟩ ≡ 972
```
Spectrum *c_transfer = arena.Alloc<Spectrum>(SHTerms(lmax));
SHComputeDiffuseTransfer(p, Faceforward(n, wo), isect.rayEpsilon,
                         scene, rng, nSamples, lmax, c_transfer);
```

Computing the SH coefficients of the transfer function at a given point can be done easily with Monte Carlo integration via Equation (17.11), similarly to how we've previously projected other functions into the SH basis.

BBox 70
Camera 302
Camera::shutterOpen 302
DiffusePRTIntegrator 972
DiffusePRTIntegrator::c_in 972
DiffusePRTIntegrator::lmax 972
DiffusePRTIntegrator::nSamples 972
Faceforward() 66
Intersection::rayEpsilon 186
MemoryArena 1015
MemoryArena::Alloc() 1016
Point 63
Renderer 24
RNG 1003
RoundUpPow2() 1002
Scene 22
Scene::WorldBound() 24
SHComputeDiffuseTransfer() 973
SHProjectIncidentDirectRadiance() 949
SHTerms() 935
Spectrum 263

⟨*Spherical Harmonics Definitions*⟩ +≡
```
void SHComputeDiffuseTransfer(const Point &p, const Normal &n,
        float rayEpsilon, const Scene *scene, RNG &rng, int nSamples,
        int lmax, Spectrum *c_transfer) {
    for (int i = 0; i < SHTerms(lmax); ++i)
        c_transfer[i] = 0.f;
    uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (int i = 0; i < nSamples; ++i) {
        ⟨Sample ith direction and compute estimate for transfer coefficients 973⟩
    }
}
```

The Monte Carlo estimate of the SH projection integral is based on uniform sampling of the sphere. The implementation here implicitly ignores transmissive surfaces by checking that the sampled direction `w` and the surface normal `n` are in the same hemisphere; a more general implementation would not perform this check if the surface was transmissive.

⟨*Sample* `i`*th direction and compute estimate for transfer coefficients*⟩ ≡ 973
```
float u[2];
Sample02(i, scramble, u);
Vector w = UniformSampleSphere(u[0], u[1]);
float pdf = UniformSpherePdf();
if (Dot(w, n) > 0.f && !scene->IntersectP(Ray(p, w, rayEpsilon))) {
    ⟨Accumulate contribution of direction ω to transfer coefficients 973⟩
}
```

If the visibility function is nonzero, then the SH functions are evaluated for the sampled direction and the coefficients are updated using the Monte Carlo estimator. A more efficient approach to implementing this function that would be appropriate for many applications would be to precompute an array of sampled directions as well as the corresponding values of the SH basis functions for each of the directions, reusing these values each time this function was called rather than calling `UniformSampleSphere()` and `SHEvaluate()` each time.

⟨*Accumulate contribution of direction ω to transfer coefficients*⟩ ≡ 973
```
SHEvaluate(w, lmax, Ylm);
for (int j = 0; j < SHTerms(lmax); ++j)
    c_transfer[j] += (Ylm[j] * AbsDot(w, n)) / (pdf * nSamples);
```

The final fragment implements the actual shading calculation, applying Equation (17.20) to compute the integral of the transfer function and the incident radiance function. It scales the result by the diffuse reflectance of the surface.

⟨*Compute integral of product of incident radiance and transfer function*⟩ ≡ 972
```
Spectrum Kd = bsdf->rho(wo, rng, BSDF_ALL_REFLECTION) * INV_PI;
Spectrum Lo = 0.f;
for (int i = 0; i < SHTerms(lmax); ++i)
    Lo += c_in[i] * c_transfer[i];
return L + Kd * Lo.Clamp();
```

AbsDot() 61
ALLOCA() 1009
BSDF::rho() 482
BSDF_ALL_REFLECTION 428
INV_PI 1002
Normal 65
Point 63
Ray 66
RNG 1003
RNG::RandomUInt() 1003
Sample02() 372
Scene 22
Scene::IntersectP() 24
SHEvaluate() 936
SHTerms() 935
Spectrum 263
Spectrum::Clamp() 265
UniformSampleSphere() 664
UniformSpherePdf() 664
Vector 57

17.5 PRECOMPUTED GLOSSY TRANSFER

The restriction to diffuse reflection imposed by the diffuse PRT approach of the previous section is a significant one; most interesting surfaces exhibit some form of directionally varying reflection. This section describes a more complex PRT method that supports more general BSDFs than Lambertian; because it is based on using a relatively small number of SH bands to represent illumination and reflection, it works well only with relatively low-frequency reflection functions. This method is more computationally intensive than diffuse PRT, though it is still practical for interactive rendering with modern graphics processors. Discussing the implementation of this approach also makes it possible to introduce a number of useful additional techniques for using spherical harmonics in rendering.

The PRT algorithm implemented in this section goes through three steps to transform the coefficients that describe the incident radiance function to coefficients that describe the scattered radiance function at a point. When $l_{\max}$ SH bands are used, each of these steps can be represented with a $(l_{\max}+1)^2 \times (l_{\max}+1)^2$ matrix that transforms one set of SH coefficients to another. The three steps are:

1. A transfer matrix $\mathbf{T}$ converts the incident radiance function arriving at the object to the incident radiance function at the particular point being shaded, accounting for shadowing by the object's geometry.
2. A rotation matrix $\mathbf{R}$ takes these SH coefficients of the incident radiance function, which is defined in the world coordinate frame, and rotates them so that the radiance function is defined in the local coordinate system at the point, where the normal vector points in the $+z$ direction.
3. The BSDF matrix $\mathbf{B}$ transforms the incident radiance function (in local coordinates) to an outgoing radiance function (still in local coordinates) accounting for how the BSDF scatters the incident illumination.

The final coefficients describe the directional distribution of outgoing radiance at the point. Given them, we can evaluate the SH basis functions for the outgoing direction to compute the outgoing radiance. Figure 17.12 shows the Buddha model rendered with the glossy PRT algorithm implemented in this section as well as a reference image rendered with the `DirectLightingIntegrator`. As long as enough SH coefficients are used, the PRT approach gives a close match.

Thus, to do the final shading calculation, given the three matrices and the coefficients for the incident radiance c_{i}, we will compute outgoing radiance coefficients c_{o} by

$$c_{\mathrm{o}} = \mathbf{B}(\mathbf{R}(\mathbf{T}c_{\mathrm{i}})),$$

and then use Equation (17.9) to compute outgoing radiance for the direction ω_{o}:

$$L_{\mathrm{o}}(\omega_{\mathrm{o}}) = \sum_i c_{\mathrm{o},i}\, Y_i(\omega_{\mathrm{o}})$$

How to rotate functions represented with SH coefficients was discussed in Section 17.2.5; there, we also saw that rotating them with a series of sparse rotation matrices was a more efficient approach than computing a dense matrix that encoded the entire rotation. Thus, the implementation to follow will perform the rotation without explicitly computing $\mathbf{R}$.

DirectLightingIntegrator 742

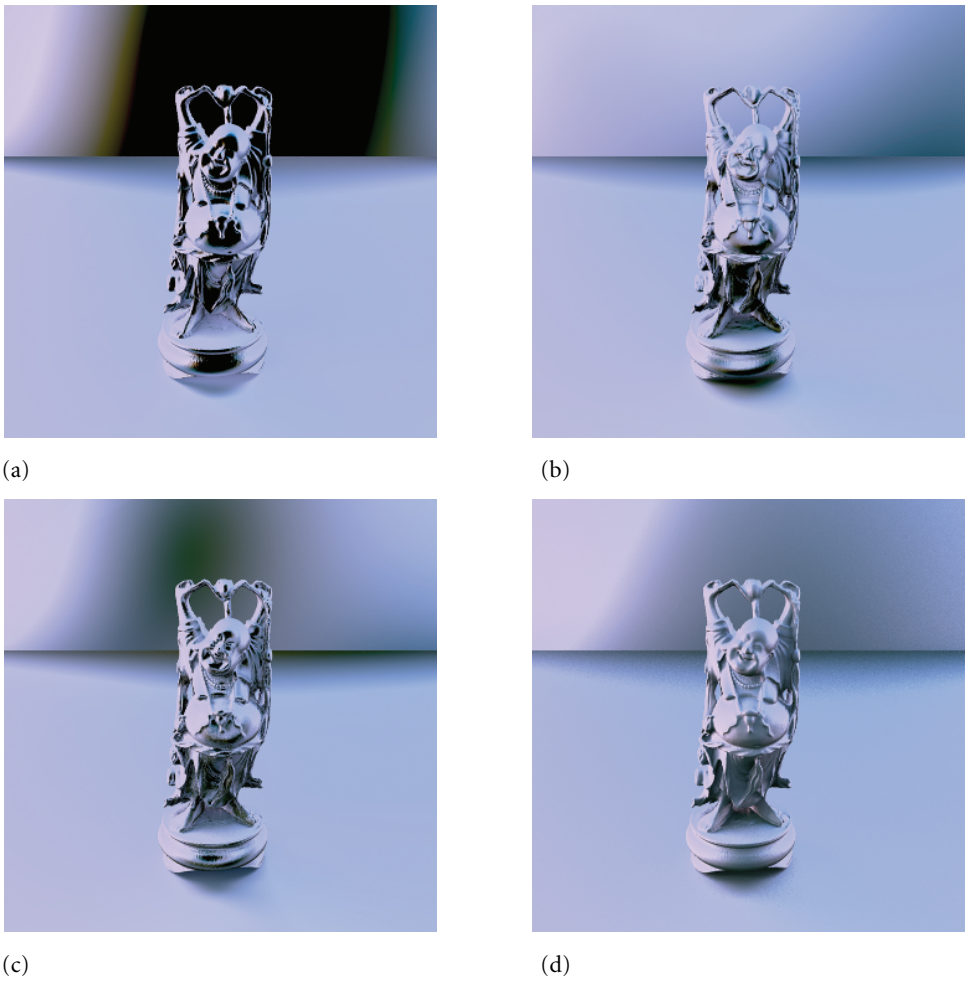


Figure 17.12: Buddha Model Rendered Using Glossy Precomputed Radiance Transfer. (a) A glossy Buddha illuminated by an infinite area light source, rendered with the `GlossyPRTIntegrator` with $l_{\max} = 3$, (b) `GlossyPRTIntegrator` with $l_{\max} = 5$, (c) `GlossyPRTIntegrator` with $l_{\max} = 8$, (d) reference image rendered with the `DirectLightingIntegrator`. The approximations introduced by representing illumination in the SH accurately render the object as long as enough SH coefficients are used.

We'll first show how the transfer and BSDF matrices are computed in the two following sections.

17.5.1 THE TRANSFER MATRIX

`DirectLightingIntegrator` 742

Given an unoccluded incident radiance function $L_{\text{env}}(\text{p}, \omega)$ (unknown until rendering time) and a transfer function $T(\text{p}, \omega)$ (which can be evaluated at precomputation time), we would like to be able to efficiently compute the SH coefficients of the directionally varying incident radiance function given by their product,

$$L_{\text{i}}(\text{p}, \omega) = L_{\text{env}}(\text{p}, \omega)\, T(\text{p}, \omega),$$

once the SH coefficients of the incident radiance function are known. Here, the transfer function is given by $T(\mathrm{p}, \omega_\mathrm{i}) = V(\mathrm{p}, \omega_\mathrm{i})$, and doesn't include the cosine term in Equation (17.19). (The cosine factor will be incorporated as part of the BSDF matrix.)

More generally, we would like to be able to multiply an unknown function represented in SH (the incident radiance function) with a known function (the transfer function) and compute the coefficients of the SH projection of the result.

If we use the indexed notation for SH coefficients from Equation (17.10), c_i are the coefficients for the incident radiance function and c'_i are the SH coefficients we'd like to compute for the product of the incident radiance and visibility functions. Application of the spherical harmonics projection formula says that the ith coefficient of c'_i is

$$c'_i = \int_{\mathbb{S}^2} \left(\sum_j c_j Y_j(\omega) \right) T(\mathrm{p}, \omega)\, Y_i(\omega)\, \mathrm{d}\omega.$$

We can interchange the integral and the sum and pull the constant c_j terms out of the integral, giving

$$c'_i = \sum_j c_j \int_{\mathbb{S}^2} Y_j(\omega)\, T(\mathrm{p}, \omega)\, Y_i(\omega)\, \mathrm{d}\omega.$$

The remaining terms in the integral can now all be evaluated at preprocessing time. They can be stored in the form of a matrix that can be used to compute new SH c'_i coefficients given coefficients c_i:

$$\begin{pmatrix} c'_0 \\ c'_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} t_{0,0} & t_{0,1} & \cdots \\ t_{1,0} & \ddots & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \end{pmatrix}$$

where

$$t_{i,j} = \int_{\mathbb{S}^2} Y_j(\omega)\, T(\mathrm{p}, \omega)\, Y_i(\omega)\, \mathrm{d}\omega. \tag{17.21}$$

Note that this matrix is symmetric.

The `SHComputeTransferMatrix()` method computes the transfer matrix for a given position in the scene, returning it in an $(l_{\max}+1)^2 \times (l_{\max}+1)^2$ array `T` allocated by the caller.

⟨*Spherical Harmonics Definitions*⟩ +≡

```
void SHComputeTransferMatrix(const Point &p, float rayEpsilon,
        const Scene *scene, RNG &rng, int nSamples, int lmax,
        Spectrum *T) {
    for (int i = 0; i < SHTerms(lmax)*SHTerms(lmax); ++i)
        T[i] = 0.f;
    uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
    float *Ylm = ALLOCA(float, SHTerms(lmax));
    for (int i = 0; i < nSamples; ++i) {
        ⟨Compute Monte Carlo estimate of i th sample for transfer matrix 977⟩
    }
}
```

Point 63
RNG 1003
RNG::RandomUInt() 1003
Scene 22
Spectrum 263

Monte Carlo integration is used to estimate the values of the integrals in Equation (17.21). Each sample ω is used to update the estimates of all of the $(l_{\max}+1)^2 \times (l_{\max}+1)^2$ integrals that need to be computed; doing so lets us reuse the result of the relatively expensive shadow ray test, improving efficiency.

⟨*Compute Monte Carlo estimate of i th sample for transfer matrix*⟩ ≡ 976

```
float u[2];
Sample02(i, scramble, u);
Vector w = UniformSampleSphere(u[0], u[1]);
float pdf = UniformSpherePdf();
if (!scene->IntersectP(Ray(p, w, rayEpsilon))) {
    ⟨Update transfer matrix for unoccluded direction 977⟩
}
```

If the transfer function for the sampled direction `w` is one, then the estimates of the integrands for all of the elements of the matrix are updated.

⟨*Update transfer matrix for unoccluded direction*⟩ ≡ 977

```
SHEvaluate(w, lmax, Ylm);
for (int j = 0; j < SHTerms(lmax); ++j)
    for (int k = 0; k < SHTerms(lmax); ++k)
        T[j*SHTerms(lmax)+k] += (Ylm[j] * Ylm[k]) / (pdf * nSamples);
```

17.5.2 THE BSDF MATRIX

Given a known BSDF, $f(\omega_o, \omega_i)$, we'd like to compute the SH coefficients for the outgoing radiance function L_o that results from integrating the product of the BSDF and $\cos\theta$ term with an incident radiance function defined in SH.

The scattering equation, Equation (5.8), describes the value to be computed in the general case; if the incident radiance function is described by SH coefficients c_j, we have

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) \left(\sum_j c_j Y_j(\omega_i) \right) |\cos\theta_i| \, d\omega_i.$$

We'd like to project the outgoing radiance function, $L_o(p, \omega_o)$, into the spherical harmonic basis to get coefficients c'_k for the outgoing radiance function. Using the SH projection equation, Equation (17.8), we have

$$\begin{aligned} c'_k &= \int_{S^2} L_o(p, \omega_o) \, Y_k(\omega_o) \, d\omega_o \\ &= \int_{S^2} \int_{S^2} f(p, \omega_o, \omega_i) \left(\sum_j c_j \, Y_j(\omega_i) \right) |\cos\theta_i| \, Y_k(\omega_o) \, d\omega_i \, d\omega_o \end{aligned}$$

The sum and the constant terms c_j can be moved outside of the integral, giving:

$$c'_k = \sum_j c_j \int_{S^2} \int_{S^2} f(\omega_o, \omega_i) \, |\cos\theta_i| \, Y_j(\omega_i) \, Y_k(\omega_o) \, d\omega_i \, d\omega_o.$$

Similar to the transfer function in Section 17.5.1, the integrand can be evaluated if the BSDF is known.

Ray 66
Sample02() 372
Scene::IntersectP() 24
SHEvaluate() 936
SHTerms() 935
UniformSampleSphere() 664
UniformSpherePdf() 664
Vector 57

We can therefore represent the values of each of these double integrals for all (j, k) pairs in a matrix $\mathbf{B}$, where the elements of the matrix $b_{j,k}$ are given by

$$b_{j,k} = \int_{S^2} \int_{S^2} f(\omega_o, \omega_i) \, |\cos \theta_i| \, Y_j(\omega_i) \, Y_k(\omega_o) \, d\omega_i \, d\omega_o. \quad [17.22]$$

The spherical harmonic coefficients of the outgoing radiance function can then be found with a matrix multiplication of $\mathbf{B}$ with the SH coefficients of the incident radiance function,

$$c_i' = \mathbf{B} \, c_i$$

The utility function `SHComputeBSDFMatrix()` computes this matrix up to a given l_{max} band, for a specific parameterized BSDF model (diffuse and glossy reflection). Generalizing the function to handle arbitrary BSDFs would be straightforward. The BSDF matrix is returned in the `B` array, which must be at least $(l_{max} + 1)^2 \times (l_{max} + 1)^2$ elements large.

```
⟨Spherical Harmonics Definitions⟩ +≡
void SHComputeBSDFMatrix(const Spectrum &Kd, const Spectrum &Ks,
        float roughness, RNG &rng, int nSamples, int lmax, Spectrum *B) {
    for (int i = 0; i < SHTerms(lmax)*SHTerms(lmax); ++i)
        B[i] = 0.f;
    ⟨Create BSDF for computing BSDF transfer matrix⟩
    ⟨Precompute directions ω and SH values for directions 978⟩
    ⟨Compute double spherical integral for BSDF matrix 979⟩
    ⟨Free memory allocated for SH matrix computation⟩
}
```

The ⟨*Create* `BSDF` *for computing BSDF transfer matrix*⟩ fragment (not included here) does some straightforward plumbing to compute a `BSDF` object, `bsdf`, that represents to the BSDF for the provided reflection parameters.

Because it has to compute a double integral over the sphere of directions, the implementation here precomputes a set of well-distributed direction vectors on the sphere and the corresponding values of the SH basis functions for each direction. These values will be reused many times in the code that estimates the integral below.

```
⟨Precompute directions ω and SH values for directions⟩ ≡ 978
float *Ylm = new float[SHTerms(lmax) * nSamples];
Vector *w = new Vector[nSamples];
uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
for (int i = 0; i < nSamples; ++i) {
    float u[2];
    Sample02(i, scramble, u);
    w[i] = UniformSampleSphere(u[0], u[1]);
    SHEvaluate(w[i], lmax, &Ylm[SHTerms(lmax)*i]);
}
```

BSDF 478
RNG 1003
RNG::RandomUInt() 1003
Sample02() 372
SHComputeBSDFMatrix() 978
SHEvaluate() 936
SHTerms() 935
Spectrum 263
UniformSampleSphere() 664
Vector 57

Now the Monte Carlo estimate of the integrals can be computed, using estimators of the form:

$$\int_{S^2}\int_{S^2} f(\omega,\omega')\,d\omega\,d\omega' \approx \frac{1}{N_i}\frac{1}{N_j}\sum_{i}^{N_i}\sum_{j}^{N_j}\frac{f(\omega_i,\omega_j)}{p(\omega_i)\,p(\omega_j)}. \quad [17.23]$$

The implementation here uses the w array to find direction samples ω_i and ω_j for each of the estimates.

⟨*Compute double spherical integral for BSDF matrix*⟩ ≡ 978
```
for (int osamp = 0; osamp < nSamples; ++osamp) {
    const Vector &wo = w[osamp];
    for (int isamp = 0; isamp < nSamples; ++isamp) {
        const Vector &wi = w[isamp];
        ⟨Update BSDF matrix elements for sampled directions 979⟩
    }
}
```

The BSDF is evaluated for each pair of directions; if its value is nonzero, then the MC estimates for all of the elements of the matrix are updated; recall from Equation (17.22) that each element of the BSDF matrix is given by an integral of the product of this term with two SH basis functions.

⟨*Update BSDF matrix elements for sampled directions*⟩ ≡ 979
```
Spectrum f = bsdf->f(wo, wi);
if (!f.IsBlack()) {
    float pdf = UniformSpherePdf() * UniformSpherePdf();
    f *= fabsf(CosTheta(wi)) / (pdf * nSamples * nSamples);
    for (int i = 0; i < SHTerms(lmax); ++i)
        for (int j = 0; j < SHTerms(lmax); ++j)
            B[i*SHTerms(lmax)+j] += f * Ylm[isamp*SHTerms(lmax)+j] *
                                        Ylm[osamp*SHTerms(lmax)+i];
}
```

Another approach to implementing this function would be to uniformly sample the outgoing direction ω_o and then use the BSDF::Sample_f() method to sample an incident direction ω_i according to the BSDF's sampling distribution. However, doing so would mean that we wouldn't be able to reuse the precomputed Ylm values for each incident direction ω_i, but would need to evaluate the SH basis functions from scratch for each sampled direction. For highly glossy or specular BSDFs, using their sampling methods would likely be more efficient, though given that spherical harmonics aren't an efficient representation for the high-frequency lighting effects of highly glossy or specular BSDFs, this approach likely isn't worthwhile in practice.

BSDF::f() 481
BSDF::Sample_f() 706
CosTheta() 426
DiffusePRTIntegrator 972
SHTerms() 935
Spectrum 263
Spectrum::IsBlack() 265
UniformSpherePdf() 664
Vector 57

17.5.3 GlossyPRTIntegrator IMPLEMENTATION

The GlossyPRTIntegrator puts the approach outlined at the start of this section together to implement the described PRT algorithm. Like the DiffusePRTIntegrator it does some parts of the computation that would normally be done at precomputation time in its Li() method so that the approach can be easily implemented in the context of an offline rendering system like pbrt. In an interactive rendering system, the stages would normally be done at preprocessing time.

The constructor takes material parameters, the number of SH bands to use, and the number of samples to take when computing the transfer matrix. Material parameters are needed here since this implementation requires that all surfaces in the scene have the same BSDF so a single BSDF matrix can be computed. The computation of the BSDF matrix is computationally intensive, such that we would prefer to not do it at all points in the scene. Other glossy PRT implementations may be willing to pay this price or could compute a set of representative BSDF matrices and then represent BSDF matrices for arbitrary BSDFs as weighted sums of them. (For example, a single BSDF matrix could be computed for diffuse reflection, assuming reflectance of one, and then its terms could be scaled by the surfaces' actual reflectances.)

⟨*GlossyPRTIntegrator Public Methods*⟩ ≡
```
GlossyPRTIntegrator(const Spectrum &kd, const Spectrum &ks,
                    float rough, int lm, int ns)
    : Kd(kd), Ks(ks), roughness(rough), lmax(lm),
      nSamples(RoundUpPow2(ns)) {
}
```

⟨*GlossyPRTIntegrator Private Data*⟩ ≡
```
const Spectrum Kd, Ks;
const float roughness;
const int lmax, nSamples;
```

Like the `DiffusePRTIntegrator`, a single incident radiance function is used for rendering: the scene's incident lighting at the center of the scene is projected into SH, ignoring visibility, and stored in the `c_in` member variable. This computation is implemented by the ⟨*Project direct lighting into SH for* `GlossyPRTIntegrator`⟩, not included here.

⟨*GlossyPRTIntegrator Method Definitions*⟩ ≡
```
void GlossyPRTIntegrator::Preprocess(const Scene *scene,
        const Camera *camera, const Renderer *renderer) {
    ⟨Project direct lighting into SH for GlossyPRTIntegrator⟩
    ⟨Compute glossy BSDF matrix for PRT 980⟩
}
```

⟨*GlossyPRTIntegrator Private Data*⟩ +≡
```
Spectrum *c_in;
Spectrum *B;
```

After the light's SH coefficients have been stored in the `c_in` member variable, the BSDF matrix is computed and stored in `B`.

⟨*Compute glossy BSDF matrix for PRT*⟩ ≡ 980
```
B = new Spectrum[SHTerms(lmax)*SHTerms(lmax)];
SHComputeBSDFMatrix(Kd, Ks, roughness, rng, 1024, lmax, B);
```

The `GlossyPRTIntegrator::Li()` method computes outgoing radiance at a point in the scene given a ray–object intersection, using the glossy PRT algorithm. We won't include the standard parts of its implementation, focusing instead on its main fragment, ⟨*Compute reflected radiance with glossy PRT at point*⟩.

Camera 302
DiffusePRTIntegrator 972
GlossyPRTIntegrator 980
GlossyPRTIntegrator::B 980
GlossyPRTIntegrator::Kd 980
GlossyPRTIntegrator::Ks 980
GlossyPRTIntegrator::roughness 980
Renderer 24
Scene 22
SHComputeBSDFMatrix() 978
Spectrum 263

As outlined above, there are four main steps. The incident radiance function is converted to incident radiance at the point being shaded, using the transfer matrix to account for visibility between the point and the environment. Then the incident radiance function is rotated to local coordinates and an outgoing radiance function is computed using the previously computed BSDF matrix. Finally, the value of the outgoing radiance function is computed for the outgoing direction given by the incident ray.

⟨*Compute reflected radiance with glossy PRT at point*⟩ ≡
 ⟨*Compute SH radiance transfer matrix at point and SH coefficients* 981⟩
 ⟨*Rotate incident SH lighting to local coordinate frame* 981⟩
 ⟨*Compute final coefficients* c_o *using BSDF matrix* 982⟩
 ⟨*Evaluate outgoing radiance function for* ω_o *and add to* L 982⟩

⟨*Compute SH radiance transfer matrix at point and SH coefficients*⟩ ≡ 981
```
Spectrum *c_t = arena.Alloc<Spectrum>(SHTerms(lmax));
Spectrum *T = arena.Alloc<Spectrum>(SHTerms(lmax)*SHTerms(lmax));
SHComputeTransferMatrix(p, isect.rayEpsilon, scene, rng, nSamples,
                        lmax, T);
SHMatrixVectorMultiply(T, c_in, c_t, lmax);
```

The SHMatrixVectorMultiply() utility function multiplies the given vector of SH coefficients by the matrix and returns the result. Its implementation is straightforward and so won't be included here.

⟨*Spherical Harmonics Declarations*⟩ +≡
```
void SHMatrixVectorMultiply(const Spectrum *M, const Spectrum *v,
                            Spectrum *vout, int lmax);
```

Now that we have the SH coefficients of the incident radiance function at the point after accounting for occlusion by the object, we now need to rotate them to the local coordinate system defined by the surface normal and tangent vectors at the point; this rotation is necessary because the BSDF matrix was computed with respect to the local coordinate system. We find the rotation matrix by determining how the coordinate system basis vectors are transformed by the local-to-world transformations and then use the SHRotate() utility function to rotate the incident radiance function to the local coordinate system.

⟨*Rotate incident SH lighting to local coordinate frame*⟩ ≡ 981
```
Vector r1 = bsdf->LocalToWorld(Vector(1,0,0));
Vector r2 = bsdf->LocalToWorld(Vector(0,1,0));
Normal nl = Normal(bsdf->LocalToWorld(Vector(0,0,1)));
Matrix4x4 rot(r1.x, r2.x, nl.x, 0,
              r1.y, r2.y, nl.y, 0,
              r1.z, r2.z, nl.z, 0,
                 0,    0,    0, 1);
Spectrum *c_l = arena.Alloc<Spectrum>(SHTerms(lmax));
SHRotate(c_t, c_l, rot, lmax, arena);
```

Given the incident radiance function's SH coefficients in c_l and the BSDF matrix, the product of the matrix and the coefficients gives the SH coefficients of the outgoing radiance function.

BSDF::LocalToWorld() 480
GlossyPRTIntegrator::
 lmax 980
Intersection::rayEpsilon 186
Matrix4x4 1021
MemoryArena::Alloc() 1016
Normal 65
SHComputeTransferMatrix() 976
SHMatrixVectorMultiply() 981
SHRotate() 953
SHTerms() 935
Spectrum 263
Vector 57

⟨*Compute final coefficients* c_o *using BSDF matrix*⟩ ≡ 981
```
Spectrum *c_o = arena.Alloc<Spectrum>(SHTerms(lmax));
SHMatrixVectorMultiply(B, c_l, c_o, lmax);
```

Finally, to compute outgoing radiance along the ray, we just need to evaluate the SH basis functions for the outgoing direction (in local coordinates).

⟨*Evaluate outgoing radiance function for* ω_o *and add to* L⟩ ≡ 981
```
Vector woLocal = bsdf->WorldToLocal(wo);
float *Ylm = ALLOCA(float, SHTerms(lmax));
SHEvaluate(woLocal, lmax, Ylm);
Spectrum Li = 0.f;
for (int i = 0; i < SHTerms(lmax); ++i)
    Li += Ylm[i] * c_o[i];
L += Li.Clamp();
```

Implementations of glossy PRT in interactive renderers generally find the product of the **B**, **R**, and **T** matrices at each point where the precomputation is being performed (mesh vertices or samples in a texture map, etc.). However, storing even a single dense SH transformation matrix for each of a large number of points requires a substantial amount of storage. Furthermore, the computational expense of even a single full SH matrix/vector multiply at each point being shaded can also be significant.

As such, a number of compression techniques have been developed that reduce both the storage and computational requirements of storing these matrices at a large number of points. The "Further Reading" section has further details. For example, Sloan et al. (2003) suggest computing all of these matrices in the pre-processing step and then applying clustered principal component analysis to compute a small number of representative matrices from the original set. Each of the original matrices is then represented as a weighted sum of the representative matrices. Thus, the expensive matrix/SH vector multiplication only needs to be performed for the representative matrices. Each time a result is needed at a point, a weighted sum of the transformed SH vectors corresponding to the matrices gives an approximation to the transformed SH vector for the outgoing radiance function at the point; this is a substantially more efficient computation.

FURTHER READING

Early work on image-based lighting design systems was done by Airey et al. (1990), Dobashi et al. (1995), Dorsey et al. (1995), Nimeroff et al. (1994), and Teo et al. (1997), who laid the groundwork for much of the subsequent work on more general modes of precomputed light transport. More recently, Pellacini et al. (2005) and Ragan-Kelley et al. (2007) described more flexible and more efficient systems for lighting design. Kristensen et al. (2005) described a lighting design system based on precomputed light transport algorithms that consequently allows changing viewpoints, and Ben-Artzi et al. (2008) developed precomputation-based algorithms for editing BRDFs in complex scenes with fixed lighting and camera positions.

Ambient occlusion was first developed by Zhukov et al. (1998), though Miller's accessibility shading also introduced related concepts (Miller 1994). Landis (2002) described

ALLOCA() 1009
BSDF::WorldToLocal() 480
GlossyPRTIntegrator::B 980
GlossyPRTIntegrator::lmax 980
SHEvaluate() 936
SHMatrixVectorMultiply() 981
SHTerms() 935
Spectrum 263
Spectrum::Clamp() 265
Vector 57

its application for offline rendering for movies, and Kontkanen and Laine (2005) and Kontkanen and Aila (2006) described ambient occlusion representations that are applicable to animated and moving objects. See also the survey article by Méndez-Feliu and Sbert (2009) for more information about techniques based on and related to ambient occlusion.

Patmore precomputed incident directional radiance on a grid for illuminating foliage (Patmore 1993), and Greger et al. (1998) introduced the *irradiance volume*, where directional irradiance functions are stored on an adaptive grid and used for illuminating objects in a scene. Both of these approaches laid the foundation for radiance probes based on spherical harmonic coefficients. Kontkanen and Laine (2006) developed methods for reducing rendering artifacts due to radiance probes being computed inside solid objects; the approach used in Section 17.3 is inspired by their insights. Nijasure et al. (2005) developed a real-time global illumination algorithm for dynamic scenes based on dynamically generated radiance probes exchanging illumination with their neighbors.

Kajiya and Von Herzen (1984) first introduced spherical harmonics to graphics. Cabral et al. (1987) used them to represent BRDFs, and BRDF matrices of the form used in Section 17.5 were used to represent complex simulated reflection functions for Monte Carlo path tracing by Westin et al. (1992). Kautz et al. (2002) tabularized spherical harmonic coefficients of 2D slices of the 4D BRDF for a set of outgoing directions and then interpolated between them, making it possible to efficiently compute glossy reflection by using the efficient inner product method of Equation (17.4). Génevaux et al. (2006) described an approach for interactive rendering of specular refraction where precomputed refraction paths are stored in the spherical harmonics basis.

Sloan's notes (Sloan 2008) and Snyder's technical report (Snyder 2006) collected a variety of useful information about using spherical harmonics in practice. Krivánek et al. (2006) developed an approximate method for rotation SH functions that is more efficient than the approach used in this chapter. For functions defined only on the hemisphere, a variant of the SH basis that only covers the hemisphere was introduced by Guatron et al. (2004). Jarosz et al. (2009) showed how to generate samples from functions represented in SH using importance sampling.

Ramamoorthi has formalized and developed the approach of expressing reflection in terms of convolution, deriving a number of important new algorithms for rendering. His Ph.D. thesis (Ramamoorthi 2002) is a valuable reference, with pointers to a series of publications in this area. See in particular his papers with Hanrahan (Ramamoorthi and Hanrahan 2001a, 2001b) for the development of the theory and its application to computing irradiance from radiance functions (which we applied in Section 17.3.2). Basri and Jacobs (2001) simultaneously derived this approach for irradiance functions.

The first modern precomputed radiance transport techniques were introduced by Sloan and collaborators (Sloan et al. 2002, 2003a, 2003b). The diffuse PRT algorithm implemented in Section 17.4 was introduced in their 2002 paper, and the glossy PRT algorithm from Section 17.5 is similar to the one in Sloan et al. (2003a), though we have not implemented the compression technique they introduced here. Other early work includes Lehtinen and Kautz (2003), who used the general BRDF matrix approach for PRT, though they also introduced an even more efficient approach for run-time

evaluation than we used here. See Lehtinen's M.S. thesis and paper for a solid mathematical framework for describing PRT algorithms in terms of standard rendering theory (Lehtinen 2004, 2007). Ramamoorthi has published an extensive survey of precomputed light transport algorithms; it is an excellent resource for more information and pointers to other work on this topic (Ramamoorthi 2007).

Since the introduction of initial PRT algorithms, a substantial amount of work has been done by researchers to increase the generality of scenes that can be rendered using PRT approaches. For example, James and Fatahalian (2003) and Sloan et al. (2005) introduced methods to support animated models, and Wang et al. (2005) applied PRT to subsurface scattering. Zhou et al. (2005) and Ren et al. (2006) developed techniques for precomputing the effect of occlusion from complex objects for more efficient real-time rendering; these techniques can also be used for computing more accurate incident radiance functions for objects when using other PRT algorithms. Annen et al. (2004) showed how SH illumination from local light sources (rather than infinitely far away light sources) can be efficiently incorporated.

A particularly effective technique for efficiently handling general BRDFs with PRT has been to use *separable approximations*. Kautz and McCool developed a (non-PRT-based) approach for interactive rendering that approximated 4D BRDFs $f(\omega_o, \omega_i)$ as products of two 2D functions $g(\omega_i)h(\omega_o)$ (Kautz and McCool 1999). This approach was first used for PRT by Liu et al. (2004) and Wang et al. (2004). Mahajan et al. (2008) analyzed the error from these factorization approaches and provided pointers to other recent applications of them.

One disadvantage of the spherical harmonics basis is that it can't represent sharp reflections and shadows accurately. Ng et al. (2003) demonstrated that non-linear approximation using wavelets could effectively capture these all-frequency lighting effects for PRT. Wang et al. (2006b) applied wavelets and a separable BRDF approximation to rendering glossy objects, and Wang et al. (2006a) introduced an algorithm for rotating functions in wavelet bases; wavelets don't rotate as simply as spherical harmonics do.

For many desirable applications of PRT, it is necessary to be able to compute a *triple product* integral, the integral of the product of three functions represented in a basis. (For example, we might have an incident radiance function, a BRDF function, and a visibility function all represented in a particular basis.) Unfortunately, computing integrals of triple products of functions is much less efficient than the double product of Equation (17.4). Ng et al. (2004) investigated triple product integrals in depth and presented a new $O(n)$ algorithm for triple products for functions in the Haar wavelet basis, with n the number of basis function coefficients. (In contrast, triple product integrals for spherical harmonics require $O(n^{5/2})$ operations.)

EXERCISES

2 17.1 Use `pbrt` to compute radiance probes for a scene and write an interactive rendering system that loads these probes and uses them for shading dynamic objects. Experiment with computing probes at different sampling frequencies;

how fine a spacing is necessary for artifact-free rendering? What sort of visual errors result from too low a sampling frequency?

● 17.2 Read Ramamoorthi and Hanrahan's paper (2004), which includes closed-form expressions for projecting a variety of symmetric BRDFs into spherical harmonics, thus making it possible to use them in the convolution framework introduced in Section 17.3.2. Implement these functions and use them to develop an extended radiance probe integrator that supports a wider variety of materials.

● 17.3 Generalize the radiance probe approach for volume light transport with multiple scattering. First, modify the probe computation process to account for volume light transport and scattering, taking these factors into account when computing the incident radiance function at a point. Next, modify the rendering process to do ray marching through `VolumeRegions` and use the radiance probes to compute scattering.

The Henyey–Greenstein phase function can be projected into spherical harmonics in closed-form; for a given asymmetry parameter g, the coefficients are given by:

$$c_l^0 = 2g^l\sqrt{(2l+1)\pi},$$

with $c_l^m = 0$ for $m \neq 0$. Use these coefficients to apply the convolution approach to computing reflected light at points in the volume.

● 17.4 Implement an interactive rendering system that uses `pbrt` to precompute PRT coefficients and then uses them for hardware-accelerated real-time rendering. You may want to implement a new `Renderer` that takes as input a number of points at which to precompute coefficients (for example, the vertices of a triangle mesh), and then stores the computed values in a file for use by the real-time renderer.

● 17.5 Read Sloan et al.'s paper on precomputed radiance transfer (Sloan et al. 2002) and modify the computation in the `DiffusePRTIntegrator` to account for multiple scattering rather than just direct lighting in the transfer function, as the current implementation does. Render images demonstrating the differences from including this factor.

● 17.6 A shortcoming of using a single incident radiance function from an infinitely far away light source for lighting in PRT algorithms is that occlusion due to other objects in the scene that weren't included in the computation of the transfer function isn't accounted for. A more accurate approach is to compute customized incident radiance functions that account for occlusion—for example, for each object in the scene.

`DiffusePRTIntegrator` 972
`Renderer` 24
`VolumeRegion` 587

Ren et al. (2006) showed how to approximate complex shadow casting objects as collections of spheres, projecting the spherical blockers into the spherical harmonic basis. They then showed how to efficiently compute products of functions in the SH basis (incident lighting and blocker occlusion) by taking the

SH logarithm, adding, and then performing SH exponentiation, similar to how multiplication of real numbers can be converted into addition via logarithms and exponentiation. (The mathematics for SH logs and exponents are much more complex, however.)

Implement an approach like Ren et al.'s and show the image differences that result from accounting for shadowing of incident radiance. What is the resulting computational overhead?

3 17.7 The glossy PRT approach in Section 17.5 is relatively inefficient in both memory and computation: dense SH matrices are computed at each point and a number of matrix multiplications are performed. Sloan et al. (2003b) introduced a compression technique that addresses both the storage and computation problems. In a preprocess, at each point where PRT computation is to be performed, the product of the transfer, rotation, and BRDF matrices is computed and stored. Then, using clustered principal component analysis (CPCA), a relatively small number of representative matrices is computed based on the initial set; each original matrix is then represented as a weighted sum of the matrices. At rendering time, the expensive matrix product with the incident radiance function is only performed for the representative matrices. Shading is just a matter of computing outgoing radiance SH coefficients as a weighted sum of the coefficients computed from the representative matrices and using these coefficients to compute the outgoing radiance value.

Implement this approach, either in a modified `GlossyPRTIntegrator` or in a hybrid system that uses `pbrt` for precomputation and a real-time renderer for rendering. Measure the difference in computation time and storage requirements. For your test scene, how many representative matrices are needed for good results? What sort of artifacts do you see with too few matrices?

GlossyPRTIntegrator 980

This page intentionally left blank

CHAPTER EIGHTEEN

18 RETROSPECTIVE AND THE FUTURE

pbrt represents one single point in the space of rendering system designs. The basic decisions we made early on—that ray tracing would be the geometric visibility algorithm used, that physical correctness would be a cornerstone of the system, and that Monte Carlo would be the main approach used for numerical integration—all had pervasive implications for the system's design. An entirely different set of trade-offs would have been made if pbrt were a renderer designed instead for real-time performance or for maximum flexibility for artistic expression. This chapter looks back at some of the details of the complete system, discusses some design alternatives, and also discusses some potential major extensions to the system that are more complex than have been described in the exercises.

18.1 DESIGN RETROSPECTIVE

One of the basic assumptions in pbrt's design was that the most interesting types of images to render are images with complex geometry and lighting. We also assumed that rendering these images well—with good sampling patterns, ray differentials, and anti-aliased textures—is worth the computational expense. One result of these assumptions is that pbrt is relatively inefficient at rendering simple images.

For example, on a 2.5-GHz dual-core CPU, pbrt takes 0.4 seconds to render a 512×512 image with no geometry or lights in the scene—a black image! 25% of this time is generating pseudo-random numbers and low-discrepancy sample points. Simple scenes with a few dozen geometric primitives and a small number of lights do not take much longer than this, indicating that the fixed per-ray cost for these parts of the computation dwarfs the expense of computing intersections and shading for trivial scenes. All of the effort used to compute samples and ray differentials for camera rays and to add the

filtered contribution of the rays to the image is clearly a substantial fraction of the time spent rendering simple scenes.

For a more complex scene, the time spent finding ray intersections, evaluating textures, and applying Monte Carlo integration algorithms dominates running time, and the relative time spent on sample generation and image sample filtering becomes less important. (For a rendering of the TT car scene, for example, pseudo-random number and sample point generation consumed 7% of run time.) Because we believe that complex scenes are the most important (and interesting) types of scenes to render, the fact that some design decisions in the system hurt performance for simple scenes was never considered a pressing problem to address.

Another performance implication of our design approach is that finding the BSDF at a ray intersection is more computationally intensive than it is in renderers that don't expend as much effort filtering textures and computing ray differentials. We believe that this effort pays off overall by reducing the need to trace more camera rays to address texture aliasing, although, again, for simple scenes, texture aliasing is often not a problem. `pbrt` generally performs better as more samples are taken to compute exitant radiance at a given intersection point. For example, increasing the number of shadow rays traced to an area light source amortizes the antialiasing work done in computing the BSDF's textures, while tracing more camera rays to reduce noise in shadows is less efficient, since a BSDF needs to be computed at each of their intersection points.

The simplicity of some of the interfaces in the system can lead to unnecessary work being done. For example, the `Sampler` always computes lens and time samples, even if they aren't needed by the `Camera`; there's no way for the `Camera` to communicate its sampling needs. Similarly, if an `Integrator` doesn't use all of the samples requested in its `GetSamples()` method for some ray, the `Sampler`'s work for generating those samples is wasted. This case can occur, for example, if the ray doesn't intersect any geometry.

Another example of potentially wasted computation is that `Shapes` always compute the partial derivatives of their normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$, even though these may not be needed. Currently, they are only used if the `BSDF` has specular components and ray differentials are being computed for the reflected or refracted rays. There's no way for the intersection routine to know if they will be needed at intersection time, however, so their values are always computed. Indeed, there's currently no way to know if the BSDF will have specular components at ray–shape intersection time. The BSDF must be created by the `Material` for this information to be known, and the `Material` needs the differential geometry at the intersection point to compute the BSDF!

One way to address this shortcoming would be to allow the material to conservatively describe all of the fields in the `DifferentialGeometry` that it needs—for example, by setting flags. These flags could be passed to the `Shape` intersection routines, which could then skip setting the member variables that aren't needed. This approach could further save execution time by allowing shapes to skip computing (u, v) parametric coordinates if they weren't needed, and so on.

BSDF 478
Camera 302
DifferentialGeometry 102
Integrator 740
Material 483
Sampler 340
Shape 108

18.1.1 ABSTRACTION VERSUS EFFICIENCY

One of the primary tensions when designing interfaces for software systems is making a reasonable trade-off between abstraction and efficiency. For example, many programmers religiously make all data in all classes `private` and provide methods to obtain or modify the values of the data items. For simple classes (e.g., `Vector`), we believe that approach needlessly hides a basic property of the implementation—that the class holds three floating-point coordinates—that we can reasonably expect to never change. Of course, using no information hiding and exposing all details of all classes' internals leads to a code maintenance nightmare. Yet, we believe that there is nothing wrong with judiciously exposing basic design decisions throughout the system. For example, the fact that a `Ray` is represented with a point, a vector, and values that give its extent, time, and recursion depth is a decision that doesn't need to be hidden behind a layer of abstraction. Code elsewhere is shorter and easier to understand when details like these are exposed.

An important thing to keep in mind when writing a software system and making these sorts of trade-offs is the expected final size of the system. The core of `pbrt` (excluding the implementations of specific shapes, lights, and so forth), where all of the basic interfaces, abstractions, and policy decisions are defined, is under 10,000 lines of code. Adding additional functionality to the system will generally only increase the amount of code in the implementations of the various abstract base classes. `pbrt` is never going to grow to be a million lines of code; this fact can and should be reflected in the amount of information hiding used in the system. It would be a waste of programmer time (and likely a source of run-time inefficiency) to design the interfaces to accommodate a system of that level of complexity.

18.1.2 DESIGN ALTERNATIVE: TRIANGLES ONLY

Another instance where the chosen abstractions impact the overall system efficiency is the range of geometric primitives that the renderer supports. While ray tracing's ability to handle a wide variety of shapes is elegant, this property is not as useful in practice as one might initially expect. Most real-world scenes are either modeled directly with polygons or with smooth surfaces like spline patches and subdivision surfaces that either have difficult-to-implement or relatively inefficient ray–shape intersection algorithms. As such, they are usually tessellated into triangles for ray intersection tests in practice. Not many shapes that are commonly encountered in real-world scenes can be described well with spheres and cones!

There are some advantages to designing a ray tracer around a single low-level shape representation like triangles and only operating on this representation throughout much of the pipeline. Such a renderer could still support a variety of primitives in the scene description but would always tessellate them at some point before performing intersection tests. Advantages of this design include the following:

- The renderer can depend on the fact that the triangle vertices can be transformed into world space in advance, so no transformations of rays into object space are ever necessary.

Ray 66
Vector 57

- The acceleration structures can be specialized so that their nodes directly store the triangles that overlap them. This improves the locality of the geometry in memory and enables ray–primitive intersection tests to be performed directly in the traversal routine, without needing to pass through two levels of virtual function calls to do so, as `pbrt` does now.
- Displacement mapping, where geometry is subdivided into small triangles, which can then have their vertices perturbed procedurally or with texture maps, can be more easily implemented if all primitives are able to tessellate themselves.

These advantages are substantial, for both increased performance and the complexity that they remove from many parts of the system. For a production renderer, rather than one with pedagogical goals like `pbrt`, this alternative is probably worth pursuing.

18.1.3 INCREASED SCENE COMPLEXITY

Given well-built acceleration structures, a strength of of ray tracing is that the time spent on ray–primitive intersections grows slowly with added scene complexity. As such, the maximum complexity that a ray tracer can handle may be limited more by memory than by computation. Because rays may pass through many different regions of the scene during a short period of time, virtual memory often performs poorly when ray tracing complex scenes due to the incoherent memory access patterns.

One way to increase the potential complexity that a renderer is capable of handling is to reduce the memory used to store the scene. For example, `pbrt` currently uses approximately 600 MB of memory for the one million triangles in the ecosystem scene in Figure 4.1. This works out to 600 bytes per triangle, if all memory use (acceleration structures, geometry, textures, and materials, etc.) is amortized over the geometric complexity. We have previously written ray tracers that managed an average of 40 bytes per triangle for scenes like these, including all memory overhead. Clearly, much reduction is possible.

Reducing memory overhead requires careful attention to memory use throughout the system. For example, in the aforementioned system, we provided three different `Triangle` implementations, one using 8-bit `uint8_t`s to store vertex indices, one using 16-bit `uint16_t`s, and one using 32-bit `uint32_t`s. The smallest index size that was sufficient for the range of vertex indices in the mesh was chosen at run time. Deering's paper on geometry compression (Deering 1995) and Ward's packed color format (Ward 1991b) are both good inspirations for thinking along these lines. See the "Further Reading" section in Chapter 4 for information about more memory-efficient acceleration structure representations.

A more complex approach is geometry caching (Pharr and Hanrahan 1996), where the renderer holds a fixed amount of geometry in memory and discards geometry that hasn't been accessed recently. This approach is useful for scenes with a lot of tessellated geometry, where a compact higher-level shape representation like a subdivision surface can explode into a large number of triangles. When available memory is low, it can be worthwhile to discard some of this geometry and regenerate it later if needed.

Triangle 139

The performance of such a cache can be substantially improved by reordering the rays that are traced in order to improve their spatial and thus memory coherence (Pharr et

al. 1997). An easier-to-implement and more effective approach to improving the cache's behavior was described by Christensen et al. (2003), who wrote a ray tracer that uses simplified representations of the scene geometry in a geometry cache. More recently, Yoon et al. (2006) and Budge et al. (2009) have developed improved approaches to this problem. See also Rushmeier, Patterson, and Veerasamy (1993) for an example of how to use simplified scene representations to compute indirect illumination.

18.2 THROUGHPUT PROCESSORS

After many years of the CPU being the heart of the computer system, recent innovations in processor design have led to a new type of high-performance processor being available in many computers. Starting with data-parallel graphics processors (GPUs) at the start of the 21st century, these new processors have made significantly more bandwidth and raw computational power available to applications than CPUs have. As of the time of writing, the hardware architectures and programming models that they support are rapidly changing. As such, this version of `pbrt` is not written to run on these new architectures. However, we will briefly discuss the development of these new processors and some topics related to implementing rendering systems like `pbrt` on them.

CPUs have long been designed to run a single thread of computation as efficiently as possible; these processors can be considered to be latency focused, in that the goal is to finish a single computation as quickly as possible. (Only since 2005 or so has this focus started to slowly change in CPU design, as multicore CPUs have provided a small number of indepdendent latency-focused processors on a single chip.) Starting with the advent of programmable graphics processors around the year 2003, *throughput processors* have increasingly become the computational heart of many computer systems. These processors focus not on single-thread performance but instead on efficiently running hundreds or thousands of computations in parallel with high aggregate computational throughput, without trying to minimize time for any of the individual computations. Major hardware vendors developing such throughput processors in 2010 include AMD, Intel, and NVIDIA.

By not focusing on single-thread performance, throughput processors are able to devote much less space on the chip for caches, branch prediction hardware, out-of-order execution units, and other features that have been invented to improve single-thread performance on CPUs. Thus, given a fixed amount of chip area, these processors are able to provide many more arithmetic logic units (ALUs) than a CPU. For the types of computation that can provide a substantial amount of independent parallel work, throughput processors can keep these ALUs busy and very efficiently execute the computation. As of the time of writing, graphics processors offer approximately ten times as many peak FLOPS as high-end CPUs; this makes them highly attractive for many processing-intensive tasks (including ray tracing).[1]

1 However, graphics processors typically consume more power and are physically larger chips than CPUs; some of their improved performance comes purely from using more power and more chip area. A fair comparison would be to consider performance per watt or per square millimeter of silicon.

Single instruction, multiple data (SIMD) processing, where processing units execute a single instruction across multiple data elements, is the key mechanism that throughput processors use to efficiently deliver computation; both today's CPUs and today's throughput processors have SIMD vector units in their processing cores. Today's multicore CPUs provide a handful of processing cores and SIMD instruction sets like SSE and AltiVec to provide four-element vector SIMD instructions on each core, while current throughput processors have tens of processing cores,[2] each with SIMD vector units between 8 and 32 elements wide. It is likely that the number of processing cores on both types of processors will go up over time, as hardware designers make use of additional transistors made possible by Moore's law. The width of the vector units is likely to change more slowly; Intel has announced an eight-element SIMD instruction set to be launched in 2010, but this is the first increase in CPU SIMD vector width on x86 processors since 1999.

For narrow SIMD widths (like four-element SSE), reasonable performance gains can be attained by opportunistically using the SIMD unit. For example, one might modify `pbrt` to use SSE instructions for the operations defined in the `Spectrum` class, thus generally being able to do three floating-point operations per instruction (for RGB spectra) rather than just one if the SIMD unit was not used. This approach would achieve 75% utilization of the SSE unit for those instructions, but doesn't help with performance in the rest of the system. In some cases, optimizing compilers can identify multiple computations in scalar code that can be executed together on SIMD units. However, achieving *excellent* utilization of SIMD vector units, especially for the SIMD vector units on throughput processors, generally requires that the entire computation be expressed in a *data parallel* manner, where the same computation is performed on many data elements simultaneously.

A natural way to extract data parallelism in a ray tracer is to have each processing core responsible for tracing n rays at a time, where n is at least the size of the SIMD width, if not larger; as such, each SIMD "lane" is responsible for just a single ray, and only a single scalar computation is performed for each of n rays at once with each instruction. Thus, high SIMD utilization comes naturally, except for the cases where some rays require different computations than others.

This approach has seen success with high-performance CPU ray tracers (where it is generally called "packet tracing"). Wald et al. (2001a) introduced this approach, which has since seen wide adoption. In a packet tracer, the camera generates "packets" of n rays which are then processed as a unit. Acceleration structure traversal algorithms are modified so that they visit a node if *any* of the rays in the packet pass through it, primitives in the leaves are tested for intersection with all of the rays simultaneously, and so forth. Packet tracing has been shown to lead to substantial speedups, although it becomes increasingly less effective as the rays to be traced become less coherent; it works well for eye rays and shadow rays to localized light sources, since the packets of rays will pass through similar regions of the scene, but efficiency generally falls off with multi-

2 The definition of a "core" on a throughput processor is notoriously tricky, with different hardware vendors promoting different definitions. Here, we are following the relatively vendor-neutral terminology proposed by Fatahalian (2008).

Spectrum 263

bounce light transport algorithms. Finding ways to retain good efficiency with packet tracing is an active area of research.

Packet tracing on CPUs is usually implemented with the SIMD vectorization made explicit: intersection functions are written to explicitly take some number of rays as a parameter rather than just a single ray, and so forth. In contrast, the vectorization in programs written for throughput processors is generally implicit: code is written as if it just operates on a single ray at a time, but the underlying compiler and hardware actually execute one instance of the program in each SIMD lane. For processors that directly expose their SIMD nature in their instruction sets (like CPUs or Intel's planned Larrabee throughput processor), the designer of the programming model is able to choose whether to provide an implicit or an explicit model to the user. See, for example, Parker et al.'s (2007) ray tracing shading language for an example of compiling an implicitly data-parallel language to a SIMD instruction set on CPUs. See also Georgiev and Slusallek's (2008) approach, where a generic programming approach is taken in C++ to allow implementing a high-performance ray tracer with details like packets well hidden.

Both the explicit and the implicit vectorization models have their advantages and disadvantages; the implicit model is generally easier for the programmer, though the explicit model allows wider visibility into the instances of computation being performed in the lanes of the vector unit, which can make more run-time optimizations possible. In either case, the programmer who cares about high performance generally must consider the effects of SIMD execution when designing algorithms and tuning performance.

18.2.1 THE FUTURE

Innovation in high-performance architectures for graphics seems likely to continue in coming years. As CPUs are gradually increasing their SIMD width and adding more processing cores, becoming more similar to throughput processors, throughput processors are adding support for task parallelism and improving their performance on more irregular workloads than purely data-parallel ones. Whether the computer system of the future is a heterogeneous collection of both types of processing cores or whether there is a middle ground with a single type of processor architecture that works well for a range of applications remains an open question.

The role of specialized fixed-function graphics hardware in future systems remains undecided as well; while recent hardware architectures have increased programmability and generality while reducing the role of fixed-function hardware, fixed-function hardware can be substantially more efficient than programmable hardware. As the critical computational kernels of future graphics systems become clear, fixed-function implementations of them may be worthwhile.

18.2.2 FURTHER RESOURCES

Purcell et al. (2002, 2003) and Carr, Hall, and Hart (2002) were the first to map general-purpose ray tracers to throughput graphics processors. While these systems do not provide many of the features available in `pbrt`, they demonstrate the feasibility of using GPUs for fast ray tracing.

Many researchers and developers have investigated the best methods for harnessing the computational power of SIMD instruction sets and throughput processors for rendering. Reshetov et al. (2005) generalized packet tracing, showing that gathering up many rays from a single origin into a frustum and then using the frustum for acceleration structure traversal could lead to very high-performance ray tracing; they refined the frusta into subfrusta and eventually the individual rays as they reached lower levels of the tree. Reshetov (2007) later introduced a technique for efficiently intersecting a collection of rays against a collection of triangles in acceleration structure leaf nodes by generating a frustum around the rays and using it for first-pass culling. See Benthin and Wald (2009) for a recent technique for using ray frusta and packets for efficient shadow rays.

Data-parallel construction of data structures tends to be challenging; see Zhou et al. (2008) and Lauterbach et al. (2009), respectively, for techniques for building kd-trees and BVHs on data-parallel throughput architectures. As throughput processors start to support task parallelism in addition to data parallelism, this task will become somewhat easier for the programmer.

The biggest current challenge in using throughput processors for rendering systems can be finding coherent collections of data-parallel computation. Consider a Monte Carlo path tracer tracing a packet of n rays; after random sampling at the first bounce, each ray will in general intersect completely different objects, likely with completely different surface shaders. At this point, running the surface shaders will likely make poor use of SIMD hardware. This specific problem of efficient shading was investigated by Hoberock et al. (2009), who resorted a large number of intersection points to make coherent collections of work before executing their surface shaders.

A number of researchers have developed methods for reordering small batches of incoherent rays to improve SIMD efficiency; representative work in this area includes papers by Mansson et al. (2007), Boulos et al. (2008), Gribble and Ramani (2008), and Tsakok (2009). Aila and Laine (2009) carefully investigated the performance of SIMD ray tracing on a graphics processor, using their insights to develop a new SIMD-friendly traversal algorithm that was substantially more efficient than the previous best known approach. Their insights are worth careful consideration for implementors of high-performance rendering systems.

An interesting trade-off for renderer developers to consider is exhibited by Hachisuka's path tracer, which uses a rasterizer with parallel projection to trace rays, effectively computing visibility in the same direction for all of the points being shaded (Hachisuka 2005). His insight was that, although this approach doesn't give a particularly good sampling distribution for Monte Carlo path tracing, in that each point isn't able to perform importance sampling to select outgoing directions, the increased efficiency from computing visibility for a very coherent collection of rays paid off overall. In other words, for a fixed amount of computation, so many more samples could be taken using rasterization versus using ray tracing that the much larger number of less well-distributed samples generated a better image than a smaller number of well-chosen samples. We suspect that this general issue of trading off between computing exactly the locally desired result at a single point versus computing what can be computed very efficiently globally for many points will be an important one for developers to consider on the throughput processors of the future.

18.3 CONCLUSION

The idea for pbrt was born in October 1999. Over the next five years, it evolved from a system designed only to support the students taking Stanford's CS348b course to a robust, feature-rich, extensible rendering system. During that time, we have both learned a great deal about what it takes to build a rendering system that doesn't just make pretty pictures, but is one that other people enjoy using and modifying as well. What has been most difficult, however, was designing a large piece of software that others would enjoy *reading*. This has been a far more challenging (and rewarding) task than implementing any of the rendering algorithms themselves.

After 2004, the book enjoyed widespread adoption in advanced graphics courses worldwide, which we have found very gratifying. We were unprepared, however, for the impact that pbrt has had on rendering research. Writing a ray tracer from scratch is a formidable task (as so many students in undergraduate graphics courses can attest to), and creating a robust physically based renderer is much harder still. We are proud that pbrt has lowered the barrier to entry for aspiring researchers in rendering, making it easier for researchers to experiment with and demonstrate the value of new ideas in rendering. Both of us continue to be delighted to see papers in SIGGRAPH, the Eurographics Rendering Symposium, High Performance Graphics, and other graphics research venues that either build on pbrt to achieve their goals, or compare their images to pbrt as "ground truth." We would like to sincerely thank everyone who has built upon this work for their own research, to build a new curriculum, or just to learn more about rendering. We hope that this new edition continues to serve the graphics community in the same way that its predecessor was able to.

This page intentionally left blank

APPENDIX A UTILITIES

In addition to all of the graphics-related code presented thus far, `pbrt` makes use of a number of general utility routines and classes. Although these are key to `pbrt`'s operation, it is not necessary to understand their implementation in detail in order to work with the rest of the system. This appendix describes the interfaces to these routines, including those that handle error reporting, memory management, the task sytem for parallel execution, and other basic infrastructure. The implementations of some of this functionality—the parts that are interesting enough to be worth delving into—are also discussed.

A.1 MAIN INCLUDE FILE

The `core/pbrt.h` file is included by all other source files in the system. It contains all global function declarations and inline functions, a few macros and numeric constants, and other globally accessible data. All files that include `pbrt.h` get a number of other included files in the process. This simplifies creation of new source files, almost all of which will want access to these extra headers. However, in the interest of compile time efficiency, we keep the number of these automatically included files to a minimum; the ones here are necessary for almost all modules.

⟨*Global Include Files*⟩ ≡

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

We also include files from the C++ standard library to get a few C++ classes that are used frequently. The `using` directive brings these classes into the default namespace.

⟨*Global Include Files*⟩ +≡
```
#include <string>
using std::string;
#include <vector>
using std::vector;
```

We will also define a macro that holds pbrt's current version number; this is used when printing the startup message in the main() function.

⟨*Global Constants*⟩ ≡
```
#define PBRT_VERSION "2.0.0"
```

A.1.1 UTILITY FUNCTIONS

A few short mathematical functions are useful throughout pbrt.

Linear Interpolation

Lerp() performs linear interpolation between two values v1 and v2, with the position given by the t parameter. When t is zero, the result is v1, and when t is one, the result is v2.

⟨*Global Inline Functions*⟩ +≡
```
inline float Lerp(float t, float v1, float v2) {
    return (1.f - t) * v1 + t * v2;
}
```

Notice that Lerp() is implemented as $(1-t)v_1 + tv_2$, rather than the more terse and potentially more computationally efficient form of $v_1 + t(v_2 - v_1)$. This is done to reduce floating-point error. If the magnitudes of v_1 and v_2 are substantially different, the difference $v_2 - v_1$ may have a substantial amount of floating-point roundoff error. When the resulting floating-point value is scaled by t and added to v_1, the result may be quite inaccurate. With the first formulation, not only is this problem avoided, but Lerp() returns *exactly* the values v_1 and v_2 when pos has values 0 and 1, respectively, and always returns a value in the range $[v_1, v_2]$ if t is in $[0, 1]$. These properties are not guaranteed by the second formulation, though other code usually at least implicitly assumes that linear interpolation will have these properties.

Clamping

Clamp() clamps the given value val to be between the values low and high:

⟨*Global Inline Functions*⟩ +≡
```
inline float Clamp(float val, float low, float high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
```

Clamp() 1000
Lerp() 1000

Modulus

`Mod()` computes the remainder of a/b. This function is handy since it behaves predictably and reasonably with negative numbers; the C and C++ standards leave the behavior of the `%` operator undefined in that case.

⟨*Global Inline Functions*⟩ +≡
```
inline int Mod(int a, int b) {
    int n = int(a/b);
    a -= n*b;
    if (a < 0) a += b;
    return a;
}
```

Converting between Angle Measures

Two simple functions convert from angles expressed in degrees to radians, and vice versa:

⟨*Global Inline Functions*⟩ +≡
```
inline float Radians(float deg) {
    return ((float)M_PI/180.f) * deg;
}
inline float Degrees(float rad) {
    return (180.f/(float)M_PI) * rad;
}
```

Base-2 Logarithms and Exponents

Because the math library doesn't provide a base-2 logarithm function, we provide one here, using the identity $\log_2(x) = \log x / \log 2$.

⟨*Global Inline Functions*⟩ +≡
```
inline float Log2(float x) {
    static float invLog2 = 1.f / logf(2.f);
    return logf(x) * invLog2;
}
```

Sometimes we need an integer-valued base-2 logarithm.

⟨*Global Inline Functions*⟩ +≡
```
inline int Log2Int(float v) {
    return Floor2Int(Log2(v));
}
```

Finally, bit manipulation techniques can be used to efficiently determine if a given integer is an exact power of two, or round an integer up to the next higher (or equal) power of two.

⟨*Global Inline Functions*⟩ +≡
```
inline bool IsPowerOf2(int v) {
    return (v & (v - 1)) == 0;
}
```

Mod() 1001
M_PI 1002

⟨*Global Inline Functions*⟩ +≡
```
inline uint32_t RoundUpPow2(uint32_t v) {
    v--;
    v |= v >> 1;    v |= v >> 2;
    v |= v >> 4;    v |= v >> 8;
    v |= v >> 16;
    return v+1;
}
```

Useful Constants

We explicitly redefine `M_PI` so that it is a 32-bit floating-point constant instead of using double precision and also define the useful constants $1/\pi$, $1/2\pi$, and $1/4\pi$.

⟨*Global Constants*⟩ +≡
```
#ifdef M_PI
#undef M_PI
#endif
#define M_PI       3.14159265358979323846f
#define INV_PI     0.31830988618379067154f
#define INV_TWOPI  0.15915494309189533577f
#define INV_FOURPI 0.07957747154594766788f
```

Finally, an `INFINITY` value is defined to be `FLT_MAX`, the largest representable floating-point number:

⟨*Global Constants*⟩ +≡
```
#ifndef INFINITY
#define INFINITY FLT_MAX
#endif
```

Floating-Point to Integer Conversion

A number of utility functions are provided for converting floating-point values to integers:

- `Float2Int(f)`: This is the same as the basic cast `(int)f`.
- `Round2Int(f)`: This rounds the floating-point value `f` to the nearest integer, returning the result as an `int`.
- `Floor2Int(f)`: The first integer value less than or equal to `f` is returned.
- `Ceil2Int(f)`: Similarly, the first integer value greater than or equal to `f` is returned.

Their default implementations are straightforward. On some architectures, these conversions can be surprisingly expensive, in which case specialized implementations can be worthwhile, especially because these functions are used in the inner loops of the system such as the texture filtering code and the film update code. See the "Further Reading" section for further details.

A.1.2 PSEUDO-RANDOM NUMBERS

`pbrt` uses a custom pseudo-random number generator rather than calling the one provided by the system. This is worth the extra effort for two reasons. First, it ensures that

INFINITY 1002
M_PI 1002

pbrt produces the same results regardless of machine architecture and C library implementation. In addition, many systems provide random number generation routines with poor statistical distributions.

The random number generator used in pbrt is the "Mersenne Twister" by Makoto Matsumoto and Takuji Nishimura. The code that implements the random number generator is both complex and subtle, and we will not attempt to explain it here. Nevertheless, it is one of the best random number generators known, can be implemented efficiently, and has a period of $2^{19937} - 1$ before it repeats the series again. A pointer to the paper describing its algorithm can be found in the "Further Reading" section.

The random number generator class, RNG, provides three methods in addition to its constructor:

- `RNG(uint32_t seed = 0)`: An optional *seed* value can be provided to the constructor; two RNGs seeded with different values will produce independent sequences of pseudo-random numbers.
- `void Seed(uint32_t seed)`: Alternatively, an RNG can be re-seeded after construction.
- `float RandomFloat() const`: A pseudo-random floating-point number in the range $[0, 1)$ is returned.
- `unsigned long RandomUInt() const`: A pseudo-random number in the range $[0, 2^{32})$ is returned.

It is defined in the files core/rng.h and core/rng.cpp.

A.2 IMAGE FILE INPUT AND OUTPUT

Many image file formats have been developed over the years, but for pbrt's purposes we are mainly interested in those that support imagery represented by floating-point pixel values. Because the images generated by pbrt will often have a large dynamic range, such formats are crucial for being able to store the computed radiance values directly; legacy image file formats, such as those that store 8 bits of data for red, green, and blue components to represent colors in the range $[0, 1]$, aren't a good fit for physically based rendering needs.

pbrt uses the OpenEXR and PFM standards as the principal file formats for images. OpenEXR is a floating-point file format designed by Industrial Light and Magic for use in their movie productions (Kainz, Bogart, and Hess 2002). A library that reads and writes this format is freely available, and support for it is available in a number of other tools. We chose this format because it has a clean design, is easy to use, and has first-class support for floating-point image data. If the preprocessor macro PBRT_HAS_OPENEXR is defined, then the OpenEXR image input and output routines are used by the system. PFM is a floating-point format based on the PPM file format; it is very easily read and written.

For convenience, pbrt also has support to read and write TGA format files. TGA is not a high-dynamic-range format like OpenEXR, but it is convenient, especially as an input format for low-dynamic-range texture maps.

pbrt provides a ReadImage() function that takes the filename to read from and pointers to two integers that will be initialized with the image resolution. It returns a freshly allocated array of RGBSpectrum objects. It will read the given file as an OpenEXR, PFM, or TGA file, depending on the suffix of the filename.

This function uses RGBSpectrum for the return values, not Spectrum. The primary client of this function is the image texture mapping code in pbrt, which stores texture maps as RGBSpectrum values, even when pbrt is compiled to do full-spectral rendering, so returning RGBSpectrum values is a natural approach. We also made this decision under the expectation that the image files being read would be in RGB or another three-channel format, so that returning RGB values wouldn't discard spectral information; if calling code wants to store full Spectrum values, then it can convert from RGB to the full-spectral representation itself, with no loss of information. If pbrt was extended to support a full-spectral image format for textures, then a variant of this function that did return Spectrum values would be necessary.

⟨*ImageIO Declarations*⟩ ≡
```
RGBSpectrum *ReadImage(const string &name, int *xSize, int *ySize);
```

The WriteImage() function takes a filename to be written, a pointer to the beginning of the pixel data, and information about the resolution of the image. The XRes and YRes variables hold the dimensions of the image to be written. If a portion of a larger image is being written, the dimensions of the larger image and the offset of the portion to be written are passed in the remaining variables. Note that these variables are not used to *save* a subimage, but rather to indicate that the pixels being passed are themselves part of a larger image. This information is written into the image header, so that the subimages can later be assembled into a single image by tools like the tools/exrassemble.cpp program in the pbrt distribution.

⟨*ImageIO Declarations*⟩ +≡
```
void WriteImage(const string &name, float *pixels, float *alpha,
    int XRes, int YRes, int totalXRes, int totalYRes, int xOffset,
    int yOffset);
```

We will not show the code that interfaces with the OpenEXR libraries or the code that implements PFM and TGA file I/O. This code can be found in the file core/imageio.cpp. The TGA implementation is based on GPL TGA code by Jaakko Keranen and Daniel Swanson; Jiawen "Kevin" Chen provided the PFM reader and writer; and the OpenEXR distribution is available from *www.openexr.com*.

A.3 COMMUNICATING WITH THE USER

The functions and classes in this section all communicate information to the user. In addition to consolidating functionality like printing progress bars, hiding user communication behind a small API like the one here also permits easy modification of the communication mechanisms. For example, if pbrt were embedded in an application that had a graphical user interface, errors might be reported via a dialog box or a routine provided by the parent application. If printf() calls were strewn throughout the system, it would be more difficult to make the two systems work together well.

ReadImage() 1004
RGBSpectrum 279
Spectrum 263
WriteImage() 1004

A.3.1 ERROR REPORTING

pbrt provides four functions for reporting anomalous conditions. In order of increasing severity, they are Info(), Warning(), Error(), and Severe(). These functions are defined in the files core/error.h and core/error.cpp. All of them take a formatting string as their first argument and a variable number of additional arguments providing values for the format. The syntax is identical to that used by the printf family of functions. For example, if the variable rayNum has type int, then the following call could be made:

```
Info("Now tracing ray number %d", rayNum);
```

core/pbrt.h includes this header file, as these functions are useful to have available in almost all parts of the system.

⟨*Global Include Files*⟩ +≡
```
#include "error.h"
```

We will not show the implementation of these functions here because they are a straightforward application of the C++ variable argument processing functions that in turn calls a common function to print the full error string. For sufficiently severe errors, the program aborts.

pbrt also has its own version of the standard assert() macro, named Assert(). It checks that the given expression's value evaluates to true; if not, Severe() is called with information about the location of the assertion failure. Assert() is used for basic sanity checks where failure indicates little possibility of recovery. In general, assertions should be used to detect internal bugs in the code, not expected error conditions (such as invalid scene file input), because the message printed will likely be cryptic to anyone other than the developer.

⟨*Global Inline Functions*⟩ +≡
```
#ifdef NDEBUG
#define Assert(expr) ((void)0)
#else
#define Assert(expr) \
    ((expr) ? (void)0 : \
        Severe("Assertion \"%s\" failed in %s, line %d", \
               #expr, __FILE__, __LINE__))
#endif // NDEBUG
```

A.3.2 REPORTING PROGRESS

Assert() 1005
ProgressReporter 1006
SamplerRenderer::Render() 27
Severe() 1005

The ProgressReporter class gives the user feedback about how much of a task has been completed and how much longer it is expected to take. For example, the implementation of the SamplerRenderer::Render() method uses a ProgressReporter to show how many of the camera rays have been traced. The current implementation prints a row of plus signs, the elapsed time, and the estimated remaining time. Its implementation is in the files core/progressreporter.h and core/progressreporter.cpp.

The constructor takes the total number of units of work to be done (e.g., the total number of camera rays that will be traced) and a short string describing the task being performed. It also takes an integer indicating the number of characters to use for the progress bar.

⟨*ProgressReporter Public Methods*⟩ ≡
```
ProgressReporter(int totalWork, const string &title,
                 int barLength = 58);
```

Once the `ProgressReporter` has been created, each call to its `Update()` method signifies that one unit of work has been completed. An optional integer value can be passed to indicate that multiple units have been done.

⟨*ProgressReporter Public Methods*⟩ +≡
```
void Update(int num = 1);
```

The `ProgressReporter::Done()` method lets the user know that the task is complete:

⟨*ProgressReporter Public Methods*⟩ +≡
```
void Done();
```

A.3.3 SIMPLE FLOAT FILE READER

A number of places in the `pbrt` code need to read simple text-format files with floating-point values. Examples include loading measured spectral data stored in text files and the measured BRDF material in Section 9.2.4. The `ReadFloatFile()` function parses simple text files of whitespace-separated numbers, returning the values found in the given `vector`. The parsing code ignores all text after a hash mark # to the end of its line to allow comments.

⟨*floatfile.h**⟩ ≡
```
bool ReadFloatFile(const char *filename, vector<float> *values);
```

A.4 PROBES AND STATISTICS

Collecting data about the run-time behavior of the system can provide a substantial amount of insight into its behavior and opportunities for improving its performance. For example, we might want to gather data to compute a histogram of how long each camera ray takes to compute a radiance value for. If some unexpectedly take much more time than others, then drilling down further may reveal a subtle bug in the system.

In support of this type of data gathering, we have defined a large number of *probes* and annotated code throughout `pbrt` to use them. The intent behind these probes is to easily make accessible relevant data from interesting points in the code's execution that we may want further information about. The programmer, in turn, can then enable a chosen set of probe points and write a small amount of code to gather the data of interest from them. Code related to these probes is defined in the files `core/probes.h` and `core/probes.cpp`.

ProgressReporter::
Done() 1006

Scene::Intersect() 23

As a concrete example, there are two probes in `Scene::Intersect()`; its actual implementation is:

```
bool Intersect(const Ray &ray, Intersection *isect) const {
    PBRT_STARTED_RAY_INTERSECTION(&ray);
    bool hit = aggregate->Intersect(ray, isect);
    PBRT_FINISHED_RAY_INTERSECTION(&ray, isect, int(hit));
    return hit;
}
```

(We have generally not included probes in the code presented in the book in the interest of brevity.) If we associate code with these probes, we can gather a number of different interesting types of information, including the total number of rays traced, the fraction of rays traced that found an intersection, the origins and directions of the rays traced (which could be useful to visualize), the average time to compute intersections for rays, and so forth.

Gathering this data can itself have a noticeable impact on performance. Therefore, `pbrt` provides a number of options for what is done with the probes in the system. In optimized builds (and more generally, when the `PBRT_PROBES_NONE` macro is defined), all of the probes disappear; all of the preprocessor macros are defined with an empty implementation. Thus, the probes have no performance impact.

⟨*Statistics Disabled Declarations*⟩ ≡

```
#define PBRT_STARTED_RAY_INTERSECTION(ray)
#define PBRT_FINISHED_RAY_INTERSECTION(ray, isect, hit)
#define PBRT_STARTED_RAY_INTERSECTIONP(ray)
#define PBRT_FINISHED_RAY_INTERSECTIONP(ray, hit)
⟨Remainder of disabled probes declarations⟩
```

A second option is provided when the `PBRT_PROBES_COUNTERS` macro is defined when the system is built. For this case, we have chosen a small number of generally interesting probes and implemented short functions that are called for them. For example, we track the total number of shapes and triangles created during rendering.

⟨*Statistics Counters Probe Definitions*⟩ ≡

```
void PBRT_CREATED_SHAPE(Shape *) {
    ++shapesMade;
}
void PBRT_CREATED_TRIANGLE(Triangle *) {
    ++trianglesMade;
}
```

These counters are tracked by simple statistics classes that associate descriptive text with the values being tracked and print out a report of results at the end of rendering. These classes also ensure that the statistics are collected in a safe manner given multithreaded execution; atomic operations are used to increment counters. Using them does cause a noticeable performance impact from having multiple threads frequently modifying the same memory location—see Section A.9.1 for discussion of this issue. A more scalable statistics system might store separate statistics for each thread in private per-thread memory and then compute aggregate results at the end of rendering.

Shape 108
Triangle 139

⟨*Statistics Counters Probe Declarations*⟩ ≡
```
static StatsCounter shapesMade("Shapes", "Total Shapes Created");
static StatsCounter trianglesMade("Shapes", "Total Triangles Created");
```

The final option for probes in `pbrt` is based on `dtrace`, a remarkable system developed by Cantrill, Shapiro, and Leventhal (2004).[1] When `pbrt` is compiled with `dtrace` support, each probe is initially converted to a small number of assembly language "no-op" instructions. When the program executes normally, these no-ops introduce a very small performance penalty—under 1% in our experience with `pbrt`'s instrumentation.

If the program is executed with data gathering enabled for some of the probes, `dtrace` modifies the executable's object code so that the no-ops for the enabled probes instead call out to data-gathering routines. These data-gathering routines are specified in a scripting language that was designed to be tailored to gathering and synthesizing performance information from complex software systems.

The `dtrace` probes corresponding to the probes above are defined as follows:

⟨*dtrace.d**⟩ ≡
```
provider PBRT {
probe started_ray_intersection(const struct Ray *);
probe finished_ray_intersection(const struct Ray *,
                                const struct Intersection *, int hit);
probe started_ray_intersectionp(const struct Ray *);
probe finished_ray_intersectionp(const struct Ray *, int hit);
⟨Remainder of dtrace probes⟩
};
```

Here is a very simple `dtrace` script; it attaches a snippet of code to the `started_ray_intersection` probe and provides another one to run when `pbrt` exits. This script counts the total number of rays that are traced and prints the total at the end.

```
uint64_t num_rays;

:::started_ray_intersection {
    ++num_rays;
}

dtrace:::END {
    printf("Total intersection tests: %d\n", num_rays);
}
```

When run on the TT car scene, this script produces the output

```
Total intersection tests: 2035434
```

Intersection 186
Ray 66

1 As of this writing, `dtrace` is available under the Mac OS X®, Solaris®, and FreeBSD® operating systems, and a port to Linux® is in progress.

dtrace transparently handles multithreaded execution, computing correct results when multiple threads simultaneously execute the code associated with a probe. dtrace scripts also provide a number of useful language features for aggregating data, producing histograms, only collecting data under certain circumstances, and so forth. For example, a relatively short script could be written to produce histograms of how many ray–triangle intersection tests were performed for rays traced for specular reflection versus those traced for shadows.

The available dtrace documentation has more information and ideas about how it can be used to investigate the behavior of complex systems. See also the example dtrace scripts provided with the pbrt distribution in the src/dtrace/ directory.

A.5 MEMORY MANAGEMENT

Memory management is often a complex issue in a system written in a language without garbage collection. The situation is mostly simple in pbrt, since most dynamic memory allocation is done as the scene description file is parsed, and most of this memory remains in use until rendering is finished. Nevertheless, there are a few issues related to memory management that warrant classes and utility routines to address them. Many of these issues are performance related, although an automatic reference-counting class is also provided to track the lifetimes of objects that may be referenced by multiple pointers in different parts of the system.

A.5.1 VARIABLE STACK ALLOCATION

Sometimes it is necessary to allocate a variable amount of memory that will be used temporarily in a single function but isn't needed after the function returns. If only a small amount of memory is needed, the overhead of new and delete (or malloc() and free()) may be high relative to the amount of actual computation being done. Instead, it is frequently more efficient to use alloca(), which efficiently allocates memory on the stack with just a few machine instructions. This memory is automatically deallocated when the function exits, which also saves bookkeeping work in the routine that uses it.

alloca() is an extremely useful tool, but there are two pitfalls to be aware of when using it. First, because the memory is deallocated when the calling routine returns, the pointer must not be returned from the routine or stored in a data structure with a longer lifetime than the function that allocated it. (However, the pointer may be passed to functions called by the allocating function.) Second, stack size is limited, and so alloca() shouldn't be used for more than a few kilobytes of storage. Unfortunately, there is no way to detect the error condition when more space is requested from alloca() than is available on the stack, so it's important to be conservative with its use.

pbrt provides a macro that makes it easy to allocate space for a given number of objects of a given type.

⟨*Global Macros*⟩ ≡

```
#define ALLOCA(TYPE, COUNT) (TYPE *)alloca((COUNT) * sizeof(TYPE))
```

A.5.2 REFERENCE-COUNTED OBJECTS

In programming languages that do not provide automatic memory management, tricky situations can arise when multiple pointers to some object exist. We would like to free an object as soon as it is no longer needed by any other object (but no sooner), so that both memory leaks and errors due to memory corruption are avoided.

As long as there aren't circular references (e.g., object A holds a reference to object B, which holds a reference to object A), a good solution to this problem is to use *reference counting*. Objects that may be pointed to in multiple places store a counter that is incremented when another object stores a reference to it and decremented when a reference goes away (e.g., due to the holding object being destroyed). When its reference count goes to zero, memory for the object can be safely freed.

We will define two classes to make it easy to use reference-counted objects in `pbrt`. An object should inherit from the `ReferenceCounted` class if it is to be managed via reference counting. The `ReferenceCounted` class adds an `nReferences` field, which is atomically incremented and decremented so that the class operates as expected when multiple threads are executing simultaneously. The count will be managed by the `Reference` class, defined in the following.

⟨*Memory Declarations*⟩ ≡
```
class ReferenceCounted {
public:
    ReferenceCounted() { nReferences = 0; }
    AtomicInt32 nReferences;
};
```

Rather than holding a pointer to a reference-counted object, an instance of the `Reference` template is used to hold the reference. The `Reference` template handles updates to the reference count as appropriate. For example, consider the following function:

```
void func() {
    Reference<Foo> r1 = new Foo;
    Reference<Foo> r2 = r1;
    r1 = new Foo;
    r2 = r1;
}
```

In the first line, a `Foo` object is allocated; `r1` holds a reference to it, and the object's `nReferences` count is one. A second reference to the object is made in the second line; `r1` and `r2` refer to the same `Foo` object, which now has a reference count of two. Next, a new `Foo` object is allocated. When a reference to it is assigned to `r1`, the reference count of the original object is decremented to one. Finally, in the last line, `r2` is assigned to refer to the newly allocated `Foo` object. The original `Foo` object now has zero references and is automatically deleted. At the end of the function, when both `r1` and `r2` go out of scope, the reference count for the second `Foo` object goes to zero, causing it to be freed as well. A few C++ language features make this all work transparently.

AtomicInt32 1036
Reference 1011
ReferenceCounted 1010

⟨*Memory Declarations*⟩ +≡
```
template <typename T> class Reference {
public:
    ⟨Reference Public Methods 1011⟩
private:
    T *ptr;
};
```

The constructors are straightforward. They just need to increment the reference count:

⟨*Reference Public Methods*⟩ ≡ 1011
```
Reference(T *p = NULL) {
    ptr = p;
    if (ptr) AtomicAdd(&ptr->nReferences, 1);
}
```

⟨*Reference Public Methods*⟩ +≡ 1011
```
Reference(const Reference<T> &r) {
    ptr = r.ptr;
    if (ptr) AtomicAdd(&ptr->nReferences, 1);
}
```

When a `Reference` is assigned to hold a different reference, it is necessary to decrement the old reference count and increment the count of the new object. The increments and decrements are ordered carefully in the following code so that an assignment like `r1 = r1` doesn't inadvertently delete the object `r1` is referring to if `r1` is the only reference.

⟨*Reference Public Methods*⟩ +≡ 1011
```
Reference &operator=(const Reference<T> &r) {
    if (r.ptr) AtomicAdd(&r.ptr->nReferences, 1);
    if (ptr && AtomicAdd(&ptr->nReferences, -1) == 0) delete ptr;
    ptr = r.ptr;
    return *this;
}
```

⟨*Reference Public Methods*⟩ +≡ 1011
```
Reference &operator=(T *p) {
    if (p) AtomicAdd(&p->nReferences, 1);
    if (ptr && AtomicAdd(&ptr->nReferences, -1) == 0) delete ptr;
    ptr = p;
    return *this;
}
```

⟨*Reference Public Methods*⟩ +≡ 1011
```
~Reference() {
    if (ptr && AtomicAdd(&ptr->nReferences, -1) == 0)
        delete ptr;
}
```

AtomicAdd() 1038
Reference 1011
Reference::ptr 1011
ReferenceCounted::
nReferences 1010

We would like to treat `References` as much like pointers as possible. For example, if the `Foo` class has a `Foo::bar()` method and `r1` is a `Reference<foo>` object, we'd like to be

able to write expressions like `r1->bar()`. The following methods take care of these details. Furthermore, the `operator bool` method makes it possible to check to see if a reference points to a `NULL` object with code like `if (!r) ...`.

⟨*Reference Public Methods*⟩ +≡ 1011

```
T *operator->() { return ptr; }
const T *operator->() const { return ptr; }
operator bool() const { return ptr != NULL; }
const T *GetPtr() const { return ptr; }
```

A.5.3 CACHE-FRIENDLY MEMORY USAGE

While the speed of modern CPUs has continued to increase at roughly the rate predicted by Moore's law (doubling every 18 to 24 months), modern memory technologies haven't been able to keep up with this growth. The speed at which memory can respond to read requests has been getting faster at a rate of roughly 10% per year. Today, a CPU will typically have to wait hundreds of execution cycles to read from main memory. The CPU is usually idle for much of this time, so a substantial amount of its computational potential may be lost.

One of the most effective techniques to address this problem is the judicious use of small, fast cache memory located either in the CPU itself or closer to it than main memory. The cache holds recently accessed data and is able to service memory requests much faster than main memory.

Because of the high penalty for accessing main memory, designing algorithms and data structures that make good use of the cache can substantially improve overall system performance. This section will discuss general programming techniques for improving cache performance. These techniques are used in many parts of `pbrt`, particularly the `KdTreeAccel`, `MIPMap`, and `ImageFilm`. We assume that the reader has a basic familiarity with computer architecture and caching technology; readers needing a review are directed to a computer architecture text such as Hennessy and Patterson (1997). In particular, the reader should be generally familiar with topics like cache lines, cache associativity, and the difference between compulsory, capacity, and conflict misses.

One easy way to reduce the number of cache misses incurred by `pbrt` is to make sure that some key memory allocations are aligned with the blocks of memory that the cache manages. Figure A.1 illustrates the basic technique. `pbrt`'s overall performance was improved by approximately 3% when allocation for the kd-tree accelerator in Section 4.5 was rewritten to use cache-aligned allocation. The `AllocAligned()` and `FreeAligned()` functions provide an interface to allocate and release cache-aligned memory blocks. If the preprocessor constant `PBRT_L1_CACHE_LINE_SIZE` is not set, a default cache line size of 64 bytes is used, which is representative of many current architectures.

⟨*Global Constants*⟩ +≡

```
#ifndef PBRT_L1_CACHE_LINE_SIZE
#define PBRT_L1_CACHE_LINE_SIZE 64
#endif
```

AllocAligned() 1013
FreeAligned() 1013
ImageFilm 404
KdTreeAccel 228
MIPMap 530
PBRT_L1_CACHE_LINE_SIZE 1012
Reference::ptr 1011

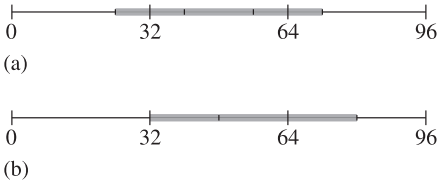


Figure A.1: The Layout of Three 16-Byte Objects in Memory on a System with 32-Byte Cache Lines. Cache-aligned memory allocation ensures that the address returned by the memory allocation routines are aligned with the start of a cache line. (a) The starting address is not cache aligned; the first and last of the three objects span two cache lines, such that two cache misses may be incurred when accessing their elements. (b) The starting address is cache aligned, guaranteeing that a maximum of one cache miss will be incurred per object.

Unfortunately there aren't portable methods to allocate memory aligned to a particular granularity. For the operating systems that do provide aligned memory allocation routines, `AllocAligned()` calls them. Otherwise, `AllocAligned()` allocates enough additional memory to ensure that the returned pointer is aligned to the desired granularity and to store the original pointer returned by `malloc()`. This pointer will be needed later to pass to `free()` when the memory is freed.

⟨*Memory Allocation Functions*⟩ ≡
```
void *AllocAligned(size_t size) {
#if defined(WIN32)
    return _aligned_malloc(size, PBRT_L1_CACHE_LINE_SIZE);
#elif defined (__OpenBSD__) || defined(__APPLE__)
    ⟨Allocate excess memory to ensure an aligned pointer can be returned⟩
#else
    return memalign(PBRT_L1_CACHE_LINE_SIZE, size);
#endif
}
```

A convenience routine is also provided for allocating a collection of objects so that code like `AllocAligned<Foo>(n)` can be written to allocate `n` instances of type `Foo`.

⟨*Memory Declarations*⟩ +≡
```
template <typename T> T *AllocAligned(uint32_t count) {
    return (T *)AllocAligned(count * sizeof(T));
}
```

The routine for freeing aligned memory either calls the operating-system-specific routine or retrieves the pointer originally returned by `malloc()` and passes that to `free()`. We won't include its implementation here.

⟨*Memory Declarations*⟩ +≡
```
void FreeAligned(void *);
```

PBRT_L1_CACHE_LINE_SIZE 1012

Another family of techniques for improving cache performance is based on reorganizing data structures themselves. For example, using bit fields to reduce the size of a frequently

used data structure can be helpful. This approach improves the *spatial locality* of memory access at run time, since code that accesses multiple packed values won't incur more than one cache miss to get them all. Furthermore, by reducing the overall size of the structure, this technique can reduce capacity misses if fewer cache lines are consequently needed to store the structure.

If not all of the elements of a structure are frequently accessed, there are a few possible strategies to improve cache performance. For example, if the structure has a size of 128 bytes and the computer has 64-byte cache lines, two cache misses may be needed to access it. If the commonly used fields are collected into the first 64 bytes rather than being spread throughout, then no more than one cache miss will be incurred when only those fields are needed (Truong, Bodin, and Seznec 1998).

A related technique is *splitting*, where data structures are split into "hot" and "cold" parts, each stored in separate regions of memory. For example, given an array of some structure type, we can split it into two arrays, one for the more frequently accessed (or "hot") portions and one for the less frequently accessed (or "cold") portions. This way, cold data doesn't displace useful information in the cache except when it is actually needed.

Cache-friendly programming is a complex engineering task, and we will not cover all the variations here. Readers are directed to the "Further Reading" section of this appendix for more information.

A.5.4 ARENA-BASED ALLOCATION

Conventional wisdom says that the system's memory allocation routines (e.g., `malloc()` and `new()`) are slow, and that custom allocation routines for objects that are frequently allocated or freed can provide a measurable performance gain. However, this conventional wisdom seems to be wrong. Wilson et al. (1995), Johnstone and Wilson (1999), and Berger, Zorn, and McKinley (2001, 2002) all investigated the performance impact of memory allocation in real-world applications and found that custom allocators almost always result in *worse* performance than a well-tuned generic system memory allocation, in both execution time and memory use.

One type of custom allocation technique that has proved to be useful in some cases is *arena-based allocation*, which allows the user to quickly allocate objects from a large contiguous region of memory. In this scheme, individual objects are never explicitly freed; the entire region of memory is released when the lifetime of all of the allocated objects ends. This type of memory allocator is a natural fit for many of the objects in `pbrt`.

There are two main advantages to arena-based allocation. First, allocation is extremely fast, usually just requiring a pointer increment. Second, it can improve locality of reference and lead to fewer cache misses, since the allocated objects are contiguous in memory. A more general dynamic memory allocator will typically prepend a bookkeeping structure to each block it returns, which adversely affects locality of reference.

`pbrt` provides the `MemoryArena` class to implement this approach; it supports variable-sized allocation from the arena.

`MemoryArena` 1015

The MemoryArena quickly allocates memory for objects of variable size by handing out pointers into a preallocated block. It does not support freeing of individual blocks of memory, only freeing of all of the memory in the zone at once. Thus, it is useful when a number of allocations need to be done quickly and all of the allocated objects have similar lifetimes.

⟨*Memory Declarations*⟩ +≡
```
class MemoryArena {
public:
    ⟨MemoryArena Public Methods 1015⟩
private:
    ⟨MemoryArena Private Data 1015⟩
};
```

MemoryArena allocates memory in chunks of size MemoryArena::blockSize, the value of which is set by a parameter passed to the constructor. It maintains a pointer to the current block of memory and the offset of the first free location in the block.

⟨*MemoryArena Public Methods*⟩ ≡ 1015
```
MemoryArena(uint32_t bs = 32768) {
    blockSize = bs;
    curBlockPos = 0;
    currentBlock = AllocAligned<char>(blockSize);
}
```

MemoryArena also uses two vectors to hold pointers to blocks of memory that have been fully used as well as available blocks that were previously allocated but aren't currently in use.

⟨*MemoryArena Private Data*⟩ ≡ 1015
```
uint32_t curBlockPos, blockSize;
char *currentBlock;
vector<char *> usedBlocks, availableBlocks;
```

To service an allocation request, the allocation routine first rounds the requested amount of memory up so that it meets the computer's word alignment requirements.[2] The routine then checks to see if the current block has enough space to handle the request, allocating a new block if necessary. Finally, it returns the pointer and updates the current block offset.

AllocAligned() 1013
MemoryArena 1015
MemoryArena::blockSize 1015
MemoryArena::curBlockPos 1015
MemoryArena::currentBlock 1015

2 Some systems (such as those based on Intel® processors) can handle non-word-aligned memory accesses, but this is usually substantially times slower than word-aligned memory reads or writes. Other architectures do not support this at all and will generate a bus error if a nonaligned access is performed.

⟨*MemoryArena Public Methods*⟩ +≡ 1015
```
void *Alloc(uint32_t sz) {
    ⟨Round up sz to minimum machine alignment 1016⟩
    if (curBlockPos + sz > blockSize) {
        ⟨Get new block of memory for MemoryArena 1016⟩
        curBlockPos = 0;
    }
    void *ret = currentBlock + curBlockPos;
    curBlockPos += sz;
    return ret;
}
```

⟨*MemoryArena Public Methods*⟩ +≡ 1015
```
template<typename T> T *Alloc(uint32_t count = 1) {
    T *ret = (T *)Alloc(count * sizeof(T));
    for (uint32_t i = 0; i < count; ++i)
        new (&ret[i]) T();
    return ret;
}
```

Most modern computer architectures impose alignment requirements on the positioning of objects in memory. For example, it is frequently a requirement that `float` values be stored at memory locations that are word aligned. To be safe, the implementation always hands out 16-byte-aligned pointers (i.e., their address is a multiple of 16).

⟨*Round up* sz *to minimum machine alignment*⟩ ≡ 1016
```
sz = ((sz + 15) & (~15));
```

If a new block of memory is needed to service request, the `MemoryArena` stores the pointer to the current block of memory on the `usedBlocks` list so that it is not lost. Later, when `MemoryArena::FreeAll()` is called, it will be able to reuse the block for the next series of allocations. The allocation routine then checks to see if there are any already allocated free blocks in the `availableBlocks` list before calling the system allocation routine to allocate a brand-new block.

⟨*Get new block of memory for* MemoryArena⟩ ≡ 1016
```
usedBlocks.push_back(currentBlock);
if (availableBlocks.size() && sz <= blockSize) {
    currentBlock = availableBlocks.back();
    availableBlocks.pop_back();
}
else
    currentBlock = AllocAligned<char>(max(sz, blockSize));
```

When the user is done with all of the memory, the arena just resets its offset in the current block and moves all of the memory from the `usedBlocks` list onto the `availableBlocks` list.

AllocAligned() 1013
MemoryArena 1015
MemoryArena::availableBlocks 1015
MemoryArena::blockSize 1015
MemoryArena::curBlockPos 1015
MemoryArena::currentBlock 1015
MemoryArena::FreeAll() 1017
MemoryArena::usedBlocks 1015

⟨*MemoryArena Public Methods*⟩ +≡ 1015

```
void FreeAll() {
    curBlockPos = 0;
    while (usedBlocks.size()) {
        availableBlocks.push_back(usedBlocks.back());
        usedBlocks.pop_back();
    }
}
```

A.5.5 BLOCKED 2D ARRAYS

In C++, 2D arrays are arranged in memory so that entire rows of values are contiguous in memory, as shown in Figure A.2(a). This is not always an optimal layout, however; for such an array indexed by (u, v), nearby (u, v) array positions will often map to distant memory locations. For all but the smallest arrays, the adjacent values in the v direction will be on different cache lines; thus, if the cost of a cache miss is incurred to reference a value at a particular location (u, v), there is no chance that handling that miss will also load into memory the data for values $(u, v + 1)$, $(u, v - 1)$, and so on. Thus, spatially coherent array indices in (u, v) do not necessarily lead to the spatially coherent memory access patterns that modern memory caches depend on.

To address this problem, the `BlockedArray` template implements a generic 2D array of values, with the items ordered in memory using a *blocked* memory layout, as shown in Figure A.2(b). The array is subdivided into square blocks of a small fixed size that is a power of two, `BLOCK_SIZE`. Each block is laid out row by row, as if it were a separate 2D C++ array. This organization substantially improves the memory coherence of 2D array references in practice and requires only a small amount of additional computation to determine the memory address for a particular position (Lam, Rothberg, and Wolf 1991).

To ensure that the block size is a power of two, the caller specifies its logarithm (base 2), which is given by the template parameter `logBlockSize`.

⟨*Memory Declarations*⟩ +≡

```
template <typename T, int logBlockSize> class BlockedArray {
public:
    ⟨BlockedArray Public Methods 1018⟩
private:
    ⟨BlockedArray Private Data 1018⟩
};
```

BlockedArray 1017
BlockedArray::RoundUp() 1019
MemoryArena::availableBlocks 1015
MemoryArena::curBlockPos 1015
MemoryArena::usedBlocks 1015

The constructor allocates space for the array and optionally initializes its values from a pointer to a standard C++ array. Because the array size may not be an exact multiple of the block size, it may be necessary to round up the size in one or both directions to find the total amount of memory needed for the blocked array. The `BlockedArray::RoundUp()` method rounds both dimensions up to be a multiple of the block size.

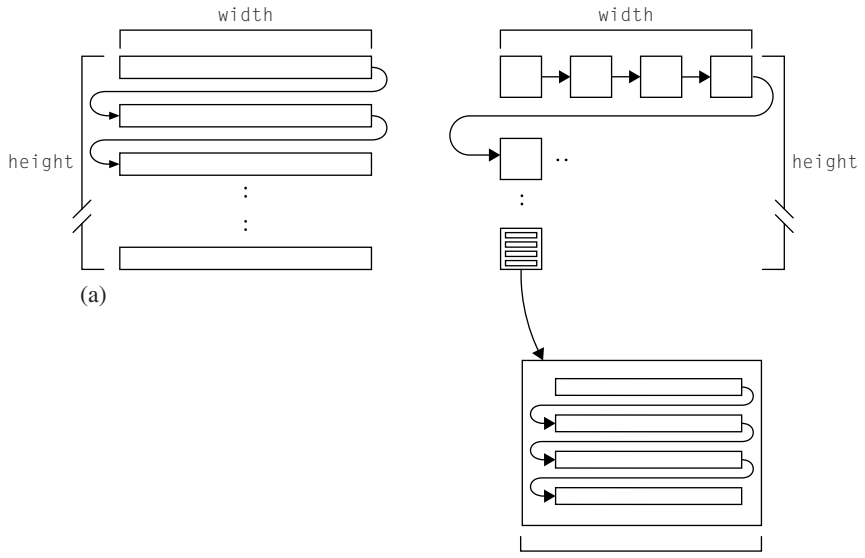


Figure A.2: (a) In C++, the natural layout for a 2D array of size `width*height` is a block of `width*height` entries, where the (u, v) array element is at the `u+v*width` offset. (b) A blocked array has been split into smaller square blocks, each of which is laid out linearly. Although it is slightly more complex to find the memory location associated with a given (u, v) array position in the blocked scheme, the improvement in cache performance due to more coherent memory access patterns often more than makes up for this in overall faster performance.

⟨*BlockedArray Public Methods*⟩ ≡ 1017

```
BlockedArray(uint32_t nu, uint32_t nv, const T *d = NULL) {
    uRes = nu;
    vRes = nv;
    uBlocks = RoundUp(uRes) >> logBlockSize;
    uint32_t nAlloc = RoundUp(uRes) * RoundUp(vRes);
    data = AllocAligned<T>(nAlloc);
    for (uint32_t i = 0; i < nAlloc; ++i)
        new (&data[i]) T();
    if (d)
        for (uint32_t v = 0; v < vRes; ++v)
            for (uint32_t u = 0; u < uRes; ++u)
                (*this)(u, v) = d[v * uRes + u];
}
```

AllocAligned() 1013
BlockedArray 1017
BlockedArray::RoundUp() 1019

⟨*BlockedArray Private Data*⟩ ≡ 1017

```
T *data;
uint32_t uRes, vRes, uBlocks;
```

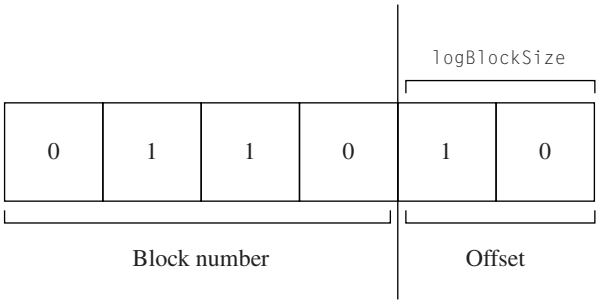


Figure A.3: Given an array coordinate, the (u, v) block number that it is in can be found by shifting off the `logBlockSize` low-order bits for both u and v. For example, with a `logBlockSize` of 2 and thus a block size of 4, we can see that this correctly maps 1D array positions from 0 to 3 to block 0, 4 to 7 to block 1, and so on. To find the offset within the particular block, it is just necessary to mask off the high-order bits, leaving the `logBlockSize` low-order bits. Because the block size is a power of two, these computations can all be done with efficient bit operations.

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
uint32_t BlockSize() const { return 1 << logBlockSize; }
uint32_t RoundUp(uint32_t x) const {
    return (x + BlockSize() - 1) & ~(BlockSize() - 1);
}
```

For convenience, the `BlockedArray` can also report its size in each dimension:

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
uint32_t uSize() const { return uRes; }
uint32_t vSize() const { return vRes; }
```

Looking up a value from a particular (u, v) position in the array requires some indexing work to find the memory location for that value. There are two steps to this process: finding which block the value is in and finding its offset within that block. Because the block sizes are always powers of two, the `logBlockSize` low-order bits in each of the u and v array positions give the offset within the block, and the high-order bits give the block number (Figure A.3).

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
uint32_t Block(uint32_t a) const { return a >> logBlockSize; }
uint32_t Offset(uint32_t a) const { return (a & (BlockSize() - 1)); }
```

BlockedArray::
BlockSize() 1019
BlockedArray::
logBlockSize 1017
BlockedArray::uRes 1018
BlockedArray::vRes 1018

Then, given the block number (b_u, b_v) and the offset within the block (o_u, o_v), it is necessary to compute what memory location this maps to in the blocked array layout. First consider the task of finding the starting address of the block; since the blocks are laid out row by row, this corresponds to the block number `bu + bv * uBlocks`, where `uBlocks` is the number of blocks in the u direction. Because each block has `BlockSize()*BlockSize()` values in it, the product of the block number and this value gives us the offset to the start

of the block. We then just need to account for the additional offset from the start of the block, which is `ou + ov * BlockSize()`.

⟨*BlockedArray Public Methods*⟩ +≡ 1017
```
T &operator()(uint32_t u, uint32_t v) {
    uint32_t bu = Block(u), bv = Block(v);
    uint32_t ou = Offset(u), ov = Offset(v);
    uint32_t offset = BlockSize() * BlockSize() * (uBlocks * bv + bu);
    offset += BlockSize() * ov + ou;
    return data[offset];
}
```

A.6 MATHEMATICAL ROUTINES

This section describes a number of useful mathematical functions and classes that support basic operations in `pbrt`, such as solving small linear systems, manipulating matrices, and linear interpolation.

A.6.1 2 × 2 LINEAR SYSTEMS

There are a number of places throughout `pbrt` where we need to solve a 2×2 linear system $Ax = B$ of the form

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}$$

for values x_0 and x_1. The `SolveLinearSystem2x2()` routine finds the closed-form solution to such a system. It returns `true` if it was successful, and `false` if the determinant of A is very small, indicating that the system is numerically ill-conditioned and either not solvable or likely to have unacceptable floating-point errors. In this case, no solution is returned.

⟨*Matrix4x4 Method Definitions*⟩ ≡
```
bool SolveLinearSystem2x2(const float A[2][2],
        const float B[2], float *x0, float *x1) {
    float det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
    if (fabsf(det) < 1e-10f)
        return false;
    *x0 = (A[1][1]*B[0] - A[0][1]*B[1]) / det;
    *x1 = (A[0][0]*B[1] - A[1][0]*B[0]) / det;
    if (isnan(*x0) || isnan(*x1))
        return false;
    return true;
}
```

BlockedArray::Block() 1019
BlockedArray::BlockSize() 1019
BlockedArray::data 1018
BlockedArray::operator() 1020
BlockedArray::uBlocks 1018
Matrix4x4 1021
Transform 76

A.6.2 4 × 4 MATRICES

The `Matrix4x4` structure provides a low-level representation of 4×4 matrices. It is an integral part of the `Transform` class.

⟨*Matrix4x4 Declarations*⟩ ≡
```
struct Matrix4x4 {
    ⟨Matrix4x4 Public Methods 1021⟩
    float m[4][4];
};
```

The default constructor, not shown here, sets the matrix to the identity matrix. It also provides constructors that allow the user to pass an array of floats or 16 individual floats to initialize a `Matrix4x4`:

⟨*Matrix4x4 Public Methods*⟩ ≡ 1021
```
Matrix4x4(float mat[4][4]);
Matrix4x4(float t00, float t01, float t02, float t03,
          float t10, float t11, float t12, float t13,
          float t20, float t21, float t22, float t23,
          float t30, float t31, float t32, float t33);
```

The implementations of operators that test for equality and inequality are straightforward and not included in the text here.

The `Matrix4x4` class supports a few low-level matrix operations. For example, `Transpose()` returns a new matrix that is the transpose of the original matrix.

⟨*Matrix4x4 Method Definitions*⟩ +≡
```
Matrix4x4 Transpose(const Matrix4x4 &m) {
   return Matrix4x4(m.m[0][0], m.m[1][0], m.m[2][0], m.m[3][0],
                    m.m[0][1], m.m[1][1], m.m[2][1], m.m[3][1],
                    m.m[0][2], m.m[1][2], m.m[2][2], m.m[3][2],
                    m.m[0][3], m.m[1][3], m.m[2][3], m.m[3][3]);
}
```

The product of two matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ is computed by setting the (i, j)th element of the result to the inner product of the ith row of $\mathbf{M}_1$ with the jth column of $\mathbf{M}_2$.

⟨*Matrix4x4 Public Methods*⟩ +≡ 1021
```
static Matrix4x4 Mul(const Matrix4x4 &m1, const Matrix4x4 &m2) {
    Matrix4x4 r;
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            r.m[i][j] = m1.m[i][0] * m2.m[0][j] +
                        m1.m[i][1] * m2.m[1][j] +
                        m1.m[i][2] * m2.m[2][j] +
                        m1.m[i][3] * m2.m[3][j];
    return r;
}
```

Inverse() 1021
Matrix4x4 1021
Matrix4x4::m 1021
Transpose() 1021

Finally, `Inverse()` returns the inverse of the matrix. The implementation (not shown here) uses a numerically stable Gauss–Jordan elimination routine to compute the inverse.

⟨*Matrix4x4 Public Methods*⟩ +≡ 1021
```
friend Matrix4x4 Inverse(const Matrix4x4 &);
```

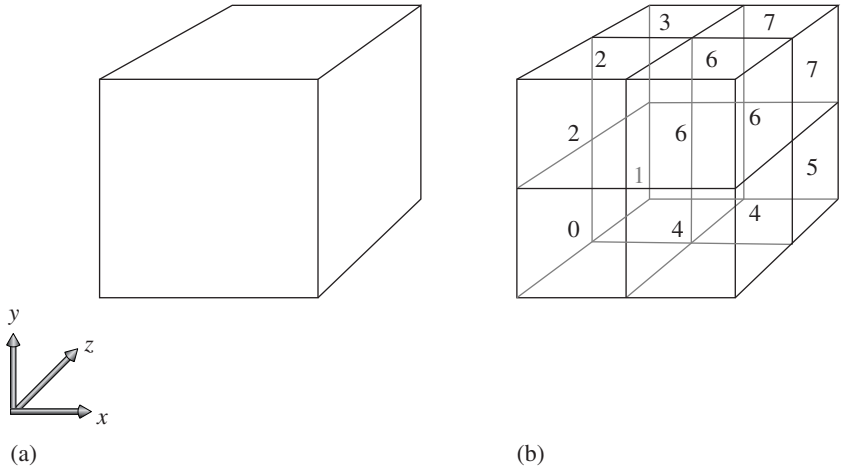


Figure A.4: Basic Octree Refinement. Starting with (a) an axis-aligned bounding box, the octree is defined by (b) progressively splitting each node into eight equal-sized child nodes. The order in which the child nodes are assigned numbers 0 to 7 is significant; details will be explained later in this section. Different subtrees may be refined to different depths, giving an adaptive discretization of 3D space.

A.7 OCTREES

An octree is a three-dimensional data structure that recursively splits a region of space into eight axis-aligned boxes (Figure A.4). The octree implementation defined in this section is in the file `core/octree.h`.

Octrees have many applications, including acceleration structures for ray tracing. The implementation in this section is specifically designed to determine which of a set of axis-aligned bounding boxes overlaps a given point. Using an octree for these queries can be substantially faster than looping over all of the objects directly. `pbrt` currently uses these octrees to store the irradiance estimates computed by the `IrradianceCacheIntegrator` and the `DipoleSubsurfaceIntegrator`. The values in these cases each have a bounding box associated with them that gives a spatial region the estimate represents.

First, we will define the `OctNode` structure, which represents a node of the tree. It holds pointers to the eight children of the node (some or all of which may be `NULL`) and a `vector` of `NodeData` objects. `NodeData` is a template argument that gives the type to be stored in the tree; for the `IrradianceCacheIntegrator`, it's the `IrradProcess` structure, which records a single irradiance estimate. The constructor and destructor of the `OctNode` just initialize the `children` to `NULL` and delete them, respectively.

DipoleSubsurfaceIntegrator 887
IrradianceCacheIntegrator 786
IrradProcess 792
NodeData 1023
OctNode 1023

⟨*Octree Declarations*⟩ ≡
```
template <typename NodeData> struct OctNode {
    OctNode() {
        for (int i = 0; i < 8; ++i)
            children[i] = NULL;
    }
    ~OctNode() {
        for (int i = 0; i < 8; ++i)
            delete children[i];
    }
    OctNode *children[8];
    vector<NodeData> data;
};
```

The `Octree` template is also parameterized by the `NodeData` type.

⟨*Octree Declarations*⟩ +≡
```
template <typename NodeData> class Octree {
public:
    ⟨Octree Public Methods 1023⟩
private:
    ⟨Octree Private Methods⟩
    ⟨Octree Private Data 1023⟩
};
```

The constructor takes the overall bounds of the tree and a maximum recursion depth beyond which nodes should not be refined:

⟨*Octree Public Methods*⟩ ≡ 1023
```
Octree(const BBox &b, int md = 16)
    : maxDepth(md), bound(b) { }
```

⟨*Octree Private Data*⟩ ≡ 1023
```
int maxDepth;
BBox bound;
OctNode<NodeData> root;
```

To add an item to the tree, the octree recurses through the tree, creating new nodes as needed, until termination criteria are met, at which point the item is added to the node it overlaps. Similar to the `KdTreeAccel` of Section 4.5, performance is highly dependent on the specific termination criteria. For example, we could decide never to refine the tree and add all items to the root node. This would be a valid octree, although it would perform poorly for large numbers of objects. However, if the tree is refined too much, items may span many nodes, leading to excessive memory use.

The `Octree::Add()` method for adding an item forwards the request to a private `Octree::addPrivate()` method with a few additional parameters, including the current node being considered, the bounding box of the node, and the squared length of the diagonal of the data item's bounding box. This method calls itself recursively as it works down the octree to the nodes where the item is stored.

BBox 70
KdTreeAccel 228
NodeData 1023
OctNode 1023
Octree 1023
Octree::Add() 1024
Octree::addPrivate() 1024
Octree::maxDepth 1023

⟨*Octree Public Methods*⟩ +≡ 1023
```
void Add(const NodeData &dataItem, const BBox &dataBound) {
    addPrivate(&root, bound, dataItem, dataBound,
               DistanceSquared(dataBound.pMin, dataBound.pMax));
}
```

The internal `Octree::addPrivate()` method either adds the item to the current node or determines which child nodes the item overlaps, allocates them if necessary, and recursively calls itself to allow the children to decide whether to add the item to their lists.

⟨*Octree Method Definitions*⟩ ≡
```
template <typename NodeData>
void Octree<NodeData>::addPrivate(
        OctNode<NodeData> *node, const BBox &nodeBound,
        const NodeData &dataItem, const BBox &dataBound,
        float diag2, int depth) {
    ⟨Possibly add data item to current octree node 1024⟩
    ⟨Otherwise add data item to octree children 1024⟩
}
```

The item is added to the current node once the maximum tree depth is reached or when the length of the diagonal of the node is less than the length of the diagonal of the item's bounds. This ensures that the item overlaps a relatively small number of tree nodes while not being too small relative to the extent of the nodes that it's added to. Figure A.5 shows the basic operation of the algorithm in two dimensions (where the corresponding data structure is known as a *quadtree*).

⟨*Possibly add data item to current octree node*⟩ ≡ 1024
```
if (depth == maxDepth ||
    DistanceSquared(nodeBound.pMin, nodeBound.pMax) < diag2) {
    node->data.push_back(dataItem);
    return;
}
```

If the method continues down the tree, it needs to determine which of the child nodes the item's bounding box overlaps. The fragment ⟨*Determine which children the item overlaps*⟩ initializes an array of Boolean values, `over[]`, such that the ith element is true only if the bounds of the data item being added overlap the ith child of the current node. The method can then loop over the eight children and recursively call `addPrivate()` for the ones that the object overlaps.

⟨*Otherwise add data item to octree children*⟩ ≡ 1024
```
Point pMid = .5 * nodeBound.pMin + .5 * nodeBound.pMax;
⟨Determine which children the item overlaps 1026⟩
for (int child = 0; child < 8; ++child) {
    if (!over[child]) continue;
    ⟨Allocate octree node if needed and continue recursive traversal 1025⟩
}
```

BBox 70
BBox::pMax 71
BBox::pMin 71
DistanceSquared() 65
NodeData 1023
OctNode 1023
Octree 1023
Octree::addPrivate() 1024
Octree::bound 1023
Octree::maxDepth 1023
Octree::root 1023
Point 63

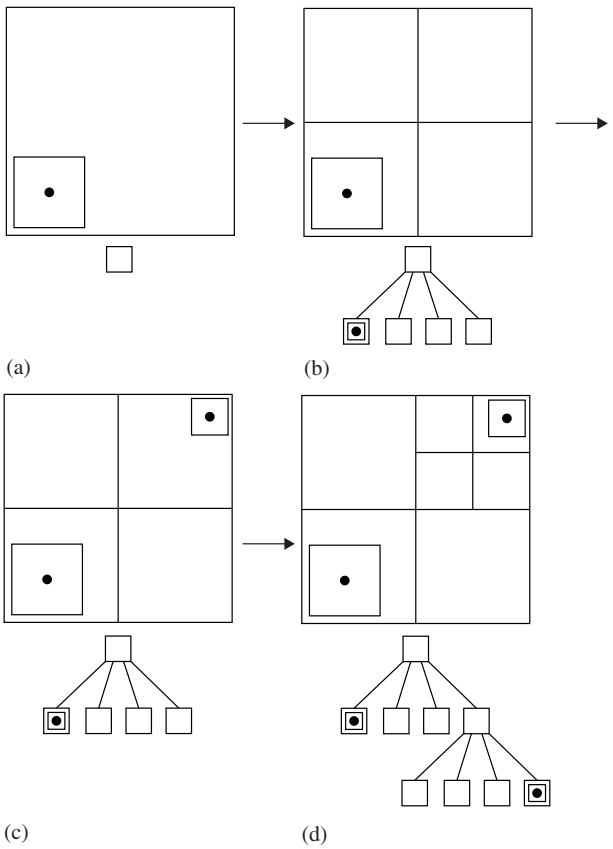


Figure A.5: Creation of a Quadtree (the 2D Analog of an Octree). (a) The tree contains just the root node, and an object with bounds around a given point is being added. The tree's topology is illustrated with a small box beneath it, corresponding to the root node with no children. (b) The tree is refined one level and the object is added to the single child node that it overlaps (again shown schematically underneath the tree). (c) Another new object with a smaller bounding box than the first is being added. (d) The tree is refined a second time before the item is added. In general, items may be stored in multiple nodes of the tree.

⟨*Allocate octree node if needed and continue recursive traversal*⟩ ≡ 1024

```
if (!node->children[child])
    node->children[child] = new OctNode<NodeData>;
BBox childBound = octreeChildBound(child, nodeBound, pMid);
addPrivate(node->children[child], childBound,
           dataItem, dataBound, diag2, depth+1);
```

BBox 70
NodeData 1023
OctNode 1023
OctNode::children 1023
Octree::addPrivate() 1024
octreeChildBound() 1026

Rather than computing the bounds of each child and doing a bounding box overlap test, it is possible to save work by taking advantage of symmetries of the situation. For example, if the x range of the object's bounding box is entirely on the left side of the plane that splits the tree node in the x direction, it cannot overlap any of the four child nodes

on the right side. Careful selection of the child node numbering scheme in Figure A.4 is the key to the success of this approach. The child nodes are numbered such that the low bit of a child's index is zero if its z component is on the low side of the z splitting plane and one if it is on the high side. Similarly, the second bit is set based on which side of the y plane the child is on, and the third bit is set based on its position with respect to the x plane. Given Boolean variables that classify a child node with respect to the splitting planes (`true` if it is above the plane), the child number of a given node is equal to

```
4 * (xHigh ? 1 : 0) + 2 * (yHigh ? 1 : 0) + 1 * (zHigh ? 1 : 0).
```

It is possible to quickly determine which child nodes a given bounding box overlaps by classifying its extent with respect to the center point of the node. For example, if the bounding box's starting x value is less than the midpoint, then the node potentially overlaps children numbers 0, 1, 2, and 3. If its ending x value is greater than the midpoint, it potentially overlaps 4, 5, 6, and 7. The following fragment checks the three dimensions in turn, computing the logical AND of the results; the item only overlaps a child node if it overlaps its extent in all three dimensions.

⟨*Determine which children the item overlaps*⟩ ≡ 1024

```
bool x[2] = { dataBound.pMin.x <= pMid.x, dataBound.pMax.x > pMid.x };
bool y[2] = { dataBound.pMin.y <= pMid.y, dataBound.pMax.y > pMid.y };
bool z[2] = { dataBound.pMin.z <= pMid.z, dataBound.pMax.z > pMid.z };
bool over[8] = { x[0] & y[0] & z[0], x[0] & y[0] & z[1],
                 x[0] & y[1] & z[0], x[0] & y[1] & z[1],
                 x[1] & y[0] & z[0], x[1] & y[0] & z[1],
                 x[1] & y[1] & z[0], x[1] & y[1] & z[1] };
```

The child node numbering scheme also makes it possible to easily find the bounding box of a particular child based on the child number and the parent node's bound:

⟨*Octree Declarations*⟩ +≡

```
inline BBox octreeChildBound(int child, const BBox &nodeBound,
                             const Point &pMid) {
    BBox childBound;
    childBound.pMin.x = (child & 4) ? pMid.x : nodeBound.pMin.x;
    childBound.pMax.x = (child & 4) ? nodeBound.pMax.x : pMid.x;
    childBound.pMin.y = (child & 2) ? pMid.y : nodeBound.pMin.y;
    childBound.pMax.y = (child & 2) ? nodeBound.pMax.y : pMid.y;
    childBound.pMin.z = (child & 1) ? pMid.z : nodeBound.pMin.z;
    childBound.pMax.z = (child & 1) ? nodeBound.pMax.z : pMid.z;
    return childBound;
}
```

After items have been added to the tree, the user can use the tree to find the items that have bounds that overlap a given point. The `Lookup()` method walks down the tree, processing the nodes that the given point overlaps. The user-supplied callback, `process`, is called for each `NodeData` item that overlaps the given point. As with the `Add()` method, the main lookup function directly calls a private version that takes a pointer to the current node and the current node's bounds.

BBox 70
NodeData 1023
Point 63

⟨*Octree Public Methods*⟩ +≡ 1023

```
template <typename LookupProc> void Lookup(const Point &p,
                                           LookupProc &process) {
    if (!bound.Inside(p)) return;
    lookupPrivate(&root, bound, p, process);
}
```

If the private lookup function has been called with a given node, the point `p` must be inside the node. The user-supplied callback is called for each `NodeData` item that is stored in the octree node, allowing the user to do whatever processing is appropriate.[3]

The callback must be either a pointer to a function that takes a position and a `NodeData` object or a class that has an `operator()` method that takes those arguments. If the callback returns `false`, traversal is halted. After the items are processed, this method continues down the tree into the single child node that `p` is inside until the bottom is reached.

⟨*Octree Method Definitions*⟩ +≡

```
template <typename NodeData> template <typename LookupProc>
bool Octree<NodeData>::lookupPrivate(OctNode<NodeData> *node,
        const BBox &nodeBound, const Point &p, LookupProc &process) {
    for (uint32_t i = 0; i < node->data.size(); ++i)
        if (!process(node->data[i]))
            return false;
    ⟨Determine which octree child node p is inside 1027⟩
    if (!node->children[child])
        return true;
    BBox childBound = octreeChildBound(child, nodeBound, pMid);
    return lookupPrivate(node->children[child], childBound, p, process);
}
```

Again taking advantage of the child numbering scheme, it is possible to quickly determine which child a point overlaps by classifying it with respect to the center of the parent node in each direction:

⟨*Determine which octree child node* `p` *is inside*⟩ ≡ 1027

```
Point pMid = .5f * nodeBound.pMin + .5f * nodeBound.pMax;
int child = (p.x > pMid.x ? 4 : 0) + (p.y > pMid.y ? 2 : 0) +
            (p.z > pMid.z ? 1 : 0);
```

BBox 70
BBox::Inside() 72
BBox::pMax 71
BBox::pMin 71
NodeData 1023
OctNode 1023
OctNode::children 1023
OctNode::data 1023
Octree 1023
Octree::bound 1023
Octree::lookupPrivate() 1027
octreeChildBound() 1026
Point 63

A.8 KD-TREES

Like the octree, the kd-tree is another data structure that accelerates the processing of spatial data. In contrast to the octree, where the data items had a known bounding box and the caller wanted to find all items that overlap a given point, the generic kd-tree

3 Note that the `Octree` actually passes all of the data items in the node to the callback, not just the subset of them that `p` is inside the bounds of. This isn't too much of a problem in practice, since the code using the `Octree` can always store a `BBox` in the `NodeData` and do the check itself. When it doesn't matter if a few extra `NodeData` items are passed back, writing the implementation in this way saves a substantial amount of `BBox` storage space.

presented in this section is useful for handling data items that are just single points in space with no associated bound, but where the caller wants to find all such points within a user-supplied distance of a given point. It is a key component of the implementation of the PhotonIntegrator.

The KdTree class described here is similar to the KdTreeAccel of Section 4.5 in that 3D space is progressively split in half by planes. There are two main differences here:

- Each tree node in the KdTree class stores a single data item. Therefore, there is exactly one kd-tree node for each data item stored in the tree.
- Because each item being stored is just a single point, items never straddle the splitting plane. Therefore, they never need to be stored on both sides of a split.

One result of these differences is that it is possible to build a perfectly balanced tree, which can improve the efficiency of lookups.

Like the KdTreeAccel, the implementation here stores all of the nodes of the tree in a single contiguous array. If a node has a left child, it will immediately follow the node in the array, and the rightChild member of KdNode gives the offset to the right child of the node, if any. rightChild will be set to a very large number if there is no right child.

To further improve the cache efficiency of the kd-tree, we will apply the cache optimization described previously of separating "hot" and "cold" data. Hot data is data that is frequently accessed while the tree is being traversed, while cold data is less frequently accessed. By splitting the kd-tree node data structure into two pieces in this way, it is possible to pack hot data close together in contiguous memory, which improves performance since more tree nodes can be packed into a single cache line. Hot data is stored in KdNode structures, which record information about a node's splitting plane and its children. The additional cold data that the user wants to associate with each node is stored in a separate array, indexed identically to the KdNode array.

⟨*KdTree Declarations*⟩ ≡

```
struct KdNode {
    void init(float p, uint32_t a) {
        splitPos = p;
        splitAxis = a;
        rightChild = (1<<29)-1;
        hasLeftChild = 0;
    }
    void initLeaf() {
        splitAxis = 3;
        rightChild = (1<<29)-1;
        hasLeftChild = 0;
    }
    ⟨KdNode Data 1029⟩
};
```

KdNode 1028
KdTree 1029
KdTreeAccel 228
PhotonIntegrator 802

⟨*KdNode Data*⟩ ≡ 1028
```
float splitPos;
uint32_t splitAxis:2;
uint32_t hasLeftChild:1, rightChild:29;
```

The KdTree is a template class that is parameterized by the type of object stored in the nodes (NodeData) and the type of callback function that is used for reporting which nodes are within a given search radius of the lookup position. The KdTree implementation requires that the NodeData class has a Point member variable called NodeData::p that gives its position.

⟨*KdTree Declarations*⟩ +≡
```
template <typename NodeData> class KdTree {
public:
    ⟨KdTree Public Methods⟩
private:
    ⟨KdTree Private Methods⟩
    ⟨KdTree Private Data 1029⟩
};
```

⟨*KdTree Private Data*⟩ ≡ 1029
```
KdNode *nodes;
NodeData *nodeData;
uint32_t nNodes, nextFreeNode;
```

Because incremental addition or removal of kd-tree nodes isn't needed in pbrt, the implementation of the KdTree is made more straightforward by having all of the data to be stored passed to the KdTree constructor. The constructor allocates all of the memory needed for the tree and the data and calls the recursive tree construction function.

⟨*KdTree Method Definitions*⟩ ≡
```
template <typename NodeData>
KdTree<NodeData>::KdTree(const vector<NodeData> &d) {
    nNodes = d.size();
    nextFreeNode = 1;
    nodes = AllocAligned<KdNode>(nNodes);
    nodeData = AllocAligned<NodeData>(nNodes);
    vector<const NodeData *> buildNodes(nNodes, NULL);
    for (uint32_t i = 0; i < nNodes; ++i)
        buildNodes[i] = &d[i];
    ⟨Begin the KdTree building process 1029⟩
}
```

AllocAligned() 1013
KdNode 1028
KdTree 1029
KdTree::nextFreeNode 1029
KdTree::nNodes 1029
KdTree::nodeData 1029
KdTree::recursiveBuild() 1030
NodeData 1023
Point 63

Tree construction is handled by the recursiveBuild() method. It takes the node number of the current node to be initialized and offsets into the array data, indicating the subset of data items [start, end) from the buildNodes array to be stored beneath this node.

⟨*Begin the KdTree building process*⟩ ≡ 1029
```
recursiveBuild(0, 0, nNodes, &buildNodes[0]);
```

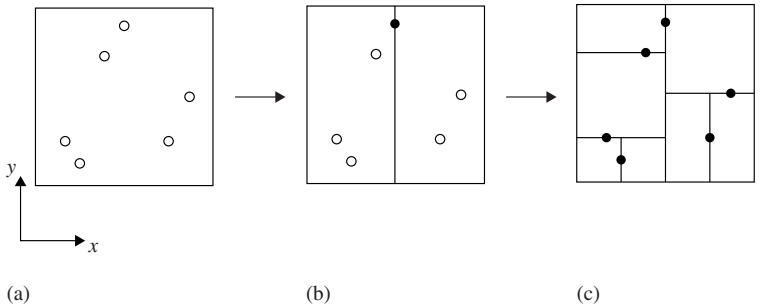


Figure A.6: Creation of a Kd-Tree to Store a Set of Points. (a) A collection of points. (b) A split direction and position are chosen. Here, we have decided to split in the x direction. We choose the point in the middle of the list sorted by x and split along the plane that goes through the point. Roughly half of points are to the left of the splitting plane and half are to the right. (c) We then continue recursively in each half, allocating new tree nodes, splitting and partitioning, until all data points have been processed.

The tree building process selects the "middle" element of the user-supplied data (what this means will soon be defined precisely) and partitions the data so that all items below the middle are in the first half of the array and all items above the middle are in the second half. It constructs a node with the middle element as its data item and then recursively initializes the two children of the node by processing the first and second halves of the array (Figure A.6).

⟨*KdTree Method Definitions*⟩ +≡

```
template <typename NodeData> void
KdTree<NodeData>::recursiveBuild(uint32_t nodeNum, int start, int end,
        const NodeData **buildNodes) {
    ⟨Create leaf node of kd-tree if we've reached the bottom 1030⟩
    ⟨Choose split direction and partition data 1031⟩
    ⟨Allocate kd-tree node and continue recursively 1031⟩
}
```

When there is just a single item to be processed, the bottom of the tree has been reached, so the node is flagged as a leaf and the `nodeData` array item at the appropriate offset is initialized from the appropriate user data.

⟨*Create leaf node of kd-tree if we've reached the bottom*⟩ ≡ 1030

```
if (start + 1 == end) {
    nodes[nodeNum].initLeaf();
    nodeData[nodeNum] = *buildNodes[start];
    return;
}
```

KdNode::initLeaf() 1028
KdTree 1029
KdTree::nodeData 1029
NodeData 1023

Otherwise, the data is partitioned into two halves, and a nonleaf node is initialized. The longest edge of the remaining data's bounding box is used to choose which axis to split along. The standard library `nth_element()` function then finds the middle node along that axis. It takes three pointers into a sequence, `start`, `mid`, and `end`, and partitions the

sequence such that the `mid`th element is in the position it would be in if the sequence were sorted. It also rearranges the array so that all elements from `start` to `mid-1` are less than `mid`, and all elements from `mid+1` to `end` are greater than `mid`. This can all be done more quickly than sorting the entire range—in $O(n)$ time rather than $O(n \log n)$.

⟨*Choose split direction and partition data*⟩ ≡ 1030
```
⟨Compute bounds of data from start to end 1031⟩
int splitAxis = bound.MaximumExtent();
int splitPos = (start+end)/2;
std::nth_element(&buildNodes[start], &buildNodes[splitPos],
                 &buildNodes[end], CompareNode<NodeData>(splitAxis));
```

It is easy to find the bounds of the entire array range that we're interested in using the `Union()` method:

⟨*Compute bounds of data from* `start` *to* `end`⟩ ≡ 1031
```
BBox bound;
for (int i = start; i < end; ++i)
    bound = Union(bound, buildNodes[i]->p);
```

The `nth_element()` function needs a "comparison object" that determines the ordering between two data elements. `CompareNode` compares positions along the chosen axis.

⟨*KdTree Declarations*⟩ +≡
```
template <typename NodeData> struct CompareNode {
    CompareNode(int a) { axis = a; }
    int axis;
    bool operator()(const NodeData *d1, const NodeData *d2) const {
        return d1->p[axis] == d2->p[axis] ? (d1 < d2) :
                                            d1->p[axis] < d2->p[axis];
    }
};
```

Once the data have been partitioned, the current node is initialized to store the middle item, and its two children are recursively initialized with the two sets of remaining items:

⟨*Allocate kd-tree node and continue recursively*⟩ ≡ 1030
```
nodes[nodeNum].init(buildNodes[splitPos]->p[splitAxis], splitAxis);
nodeData[nodeNum] = *buildNodes[splitPos];
if (start < splitPos) {
    nodes[nodeNum].hasLeftChild = 1;
    uint32_t childNum = nextFreeNode++;
    recursiveBuild(childNum, start, splitPos, buildNodes);
}
if (splitPos+1 < end) {
    nodes[nodeNum].rightChild = nextFreeNode++;
    recursiveBuild(nodes[nodeNum].rightChild, splitPos+1,
                   end, buildNodes);
}
```

BBox 70
BBox::MaximumExtent() 73
BBox::Union() 71
CompareNode 1031
KdNode::hasLeftChild 1029
KdNode::rightChild 1029
KdNode::splitAxis 1029
KdNode::splitPos 1029
KdTree::recursiveBuild() 1030
NodeData 1023
Union() 72

When another part of the system wants to look up items from the tree, it provides a point `p`, a callback procedure (similar to the one used above), and a maximum squared

search radius. Using the squared radius rather than the radius directly leads to some optimizations in the following traversal code. All data items within that radius will be passed back to the caller.

Rather than being passed by value, the squared search radius is passed into the lookup function by reference. This allows the lookup routine to pass it to the callback procedure by reference, so that the callback can reduce the search radius as the search goes on. This can speed up lookups when the callback routine can determine that a smaller search radius was appropriate after all. As usual, the lookup method immediately calls a private lookup procedure, passing in a pointer to the current node to be processed.

⟨*KdTree Method Definitions*⟩ +≡
```
template <typename NodeData> template <typename LookupProc>
void KdTree<NodeData>::Lookup(const Point &p, LookupProc &proc,
                              float &maxDistSquared) const {
    privateLookup(0, p, proc, maxDistSquared);
}
```

The lookup function has two responsibilities (Figure A.7): it needs to recursively process each of the children of the current node if they overlap the search region, and it needs to invoke the callback routine, passing it the data item in the current node if it is inside the search radius.

⟨*KdTree Method Definitions*⟩ +≡
```
template <typename NodeData> template <typename LookupProc>
void KdTree<NodeData>::privateLookup(uint32_t nodeNum, const Point &p,
        LookupProc &process, float &maxDistSquared) const {
    KdNode *node = &nodes[nodeNum];
    ⟨Process kd-tree node's children 1033⟩
    ⟨Hand kd-tree node to processing function 1033⟩
}
```

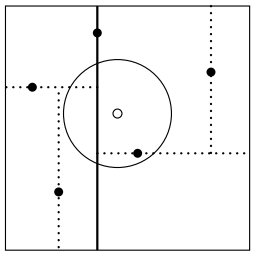


Figure A.7: Basic Process of Kd-Tree Lookups. The point marked with an open circle is the lookup position, and the region of interest is denoted by the circular region around it. At the root node of the tree (indicated by a bold splitting line), the data item is outside the region of interest, so it is not handed to the callback function. However, the region overlaps both children of the node, so we have to recursively consider each of them. In this case, the implementation here considers the right child (child number one) first in order to examine the nearby data items before examining the ones farther away.

KdNode 1028
KdTree 1029
KdTree::nodes 1029
KdTree::privateLookup() 1032
NodeData 1023
Point 63

The tree is traversed depth first, examining the leaf nodes that are close to the lookup point `p` first. This approach ensures that data points are passed to the callback function in a generally near-to-far order. If the caller is only interested in finding a fixed number of points around the lookup point, after which it will end the search, this is a more efficient order than going far to near.

Therefore, this method first recurses down the side of the tree on which the current point lies. After that lookup has returned, it checks to see if the search radius indicates that the region covers both sides of the tree. Leaf nodes are denoted by a value of 3 in the node's `splitAxis` field, in which case these steps are skipped.

⟨*Process kd-tree node's children*⟩ ≡ 1032

```
int axis = node->splitAxis;
if (axis != 3) {
    float dist2 = (p[axis] - node->splitPos) * (p[axis] - node->splitPos);
    if (p[axis] <= node->splitPos) {
        if (node->hasLeftChild)
            privateLookup(nodeNum+1, p, process, maxDistSquared);
        if (dist2 < maxDistSquared && node->rightChild < nNodes)
            privateLookup(node->rightChild, p, process, maxDistSquared);
    }
    else {
        if (node->rightChild < nNodes)
            privateLookup(node->rightChild, p, process, maxDistSquared);
        if (dist2 < maxDistSquared && node->hasLeftChild)
            privateLookup(nodeNum+1, p, process, maxDistSquared);
    }
}
```

Finally, at the end of the lookup function, a check is made to see if the point stored in the node is inside the search radius, passing it to the callback function if so. In addition to doing whatever processing it needs to do based on the item, the callback function may decrease `maxDistSquared` in order to reduce the region of space searched for the remainder of the processing.

⟨*Hand kd-tree node to processing function*⟩ ≡ 1032

```
float dist2 = DistanceSquared(nodeData[nodeNum].p, p);
if (dist2 < maxDistSquared)
    process(p, nodeData[nodeNum], dist2, maxDistSquared);
```

A.9 PARALLELISM

Section 1.3.5 introduced some basic principles of parallel programming and described their application to `pbrt`. Here, we'll provide some additional information on performance issues related to multithreading as well as the implementations of utility routines and classes for atomic memory operations, synchronization, and `pbrt`'s task system. The implementations of these routines are in the files `core/parallel.h` and `core/parallel.cpp`.

DistanceSquared() 65
KdTree::nodeData 1029

Most of this code is highly platform specific; the implementations of the atomic operations require assembly language. The forthcoming update to the C++ standard will provide thread, mutex, and atomic operations as part of the standard library, which will eliminate the need for custom implementations here.[4]

A.9.1 MEMORY COHERENCE MODELS AND PERFORMANCE

Cache coherency is a feature of all modern multicore CPUs; with it, memory writes by one processor are automatically visible to other processors. This is an incredibly useful feature; being able to assume it in the implementation of a system like `pbrt` is extremely helpful to the programmer. Understanding the subtleties and the performance characteristics of this feature is important to keep in mind, however.

One important issue is that other processors may not see writes to memory in the same order that the processor that performed the writes issued them. This can happen for two main reasons: the compiler's optimizer may have reordered write operations to improve performance, and the CPU hardware may write values to memory in a different order than the stream of executed machine instructions. In the single-threaded case, both of these are innocuous; by design, the compiler and hardware, respectively, ensure that it's impossible for a single thread of execution to detect when these cases happen. This guarantee is not provided for multithreaded code, however; doing so would impose a significant performance penalty, so hardware architectures leave handling this problem, when it matters, to software.

Memory barrier instructions can be used to ensure that all write instructions before the barrier are visible in memory before any subsequent instructions execute. In practice, we generally don't need to issue memory barrier instructions explicitly, since operating system thread synchronization calls take care of this; they are defined to do this so that if we are coordinating execution between multiple threads using these calls, then they have a consistent view of memory after synchronization points.

Although cache coherence is helpful to the programmer, it can sometimes impose a substantial performance penalty for data that is frequently modified and accessed by multiple processors. Read-only data has little penalty; copies of it can be stored in the local caches of all of the processors that are accessing it, allowing all of them the same performance benefits from the caches as in the single-threaded case. To understand the downside of taking too much advantage of cache coherence for read–write data, it's useful to understand how cache coherence is typically implemented on processors.

CPUs implement a *cache coherence protocol*, which is responsible for tracking the memory transactions issued by all of the processors in order to provide cache coherence. A widely used protocol is *MESI*, where the acronym represents the four states that each cache line can be in. Each processor stores the current state for each cache line in its local caches:

4 The Boost libraries (*www.boost.org*) provide implementations of many of these components of the new C++ standard library. We have chosen not to make `pbrt` dependent on Boost, however, in order to make it easier to build, by not requiring the user to install a number of additional libraries.

- *Modified*—The current processor has written to the memory location, but the result is only stored in the cache—it's *dirty* and hasn't been written to main memory. No other processor has the location in its cache.
- *Exclusive*—The current processor is the only one with the data from the corresponding memory location in its cache. The value in the cache matches the value in memory.
- *Shared*—Multiple processors have the corresponding memory location in their caches, but they have only performed read operations.
- *Invalid*—The cache line doesn't hold valid data.

At system startup time, the caches are empty and all cache lines are in the invalid state. If multiple threads read a memory location, the data for that location may be replicated in multiple caches, in the "exclusive" or "shared" state. If another processor performs a memory read from a location that is in the "exclusive" state in another cache, then both caches record the state for the corresponding memory location to instead be "shared."

When a processor writes to a memory location, the performance impact depends on the state of the corresponding cache line. If it's in the "exclusive" state and already in the writing processor's cache, then the write is cheap; the data is modified in the cache and the cache line's state is changed to "modified." (If it was already in the "modified" state, then the write is similarly efficient.) In these cases, the value will eventually be written to main memory, at which point the corresponding cache line returns to the "exclusive" state.

However, if a processor writes to a memory location that's in the "shared" state in its cache or is in the "modified" or "exclusive" state in another processor's cache, then expensive communication between the cores is required. All of this is handled transparently by the hardware, though it still has a performance impact. In this case, the writing processor must issue a *read for ownership* (RFO), which marks the memory location as invalid in the caches of any other processors; RFOs can cause stalls of tens or hundreds of cycles—a substantial penalty for a single memory write.

In general, we'd therefore like to avoid the situation of multiple processors concurrently writing to the same memory location as well as unnecessarily reading memory that another processor is writing to. An important case to be aware of is "false sharing," where a single cache line holds some read-only data and some data that is frequently modified. In this case, even if only a single processor is writing to the part of the cache line that is modified but many are reading from the read-only part, the overhead of frequent RFO operations will be unnecessarily incurred.

Another situation where many processors may be writing to the same memory concurrently can happen when precomputing data for rendering. For example, during the photon shooting phase in Section 15.6.3, multiple threads are concurrently tracing paths and depositing photons in the scene. Rather than simultaneously updating a single shared data structure, each thread stores photons in its own private data structure and only periodically merges its results with the global photon data structure that will actually be used for rendering. This approach allows the overhead of RFO operations to be amortized over a smaller number of larger updates.

A.9.2 ATOMIC OPERATIONS

Recall from Section 1.3.5 that mutexes can be used to safely update memory locations that are shared by multiple threads. However, when it's possible to use built-in hardware instructions for atomic updates, doing so is often more efficient than acquiring a mutex, updating the location, and releasing the mutex. Hardware atomic update instructions operate only on 4 to 8 bytes of memory at a time, however, and support only a few operations (addition, exchange, etc.). If atomic updates to more data are required, mutexes must generally be used instead.

`pbrt` defines special types for 32-bit and, if available, 64-bit values to be updated atomically. Declaring these types with the `volatile` qualifier directs the compiler to always fetch their values from memory and to not cache them in registers. This ensures that updates made by one processor are visible to others.

⟨*Parallel Declarations*⟩ ≡
```
typedef volatile int32_t AtomicInt32;
#ifdef PBRT_HAS_64_BIT_ATOMICS
typedef volatile int64_t AtomicInt64;
#endif
```

The implementation of the atomic functions are highly platform specific; the 32-bit atomic add is below. For some hardware architectures and operating systems, a library call is available to perform the atomic operation. For others, we must use assembly language.

⟨*Parallel Declarations*⟩ +≡
```
inline int32_t AtomicAdd(AtomicInt32 *v, int32_t delta) {
#ifdef WIN32
    ⟨Do atomic add with MSVC inline assembly⟩
#elif defined(__APPLE__) && !(defined(__i386__) || defined(__amd64__))
    return OSAtomicAdd32Barrier(delta, v);
#else
    ⟨Do atomic add with gcc x86 inline assembly 1037⟩
#endif
}
```

In addition to different CPU architectures having different instruction sets, we need to deal with the fact that different compilers have different syntax for specifying assembly language within C and C++ code. The fragment below uses the `xadd` x86 opcode with the `lock` modifier. (The `l` at the end of `xadd` here indicates that the value is long-word sized, which for x86 means 32 bits.)

The `lock` prefix ensures that the instruction executes as an atomic operation and `xadd` adds the parameter (here, `delta`) to the destination (the address pointed to by `v`) and returns the old value pointed to by `v`. Here, we add the value `delta` to the value returned by `xadd` and return this value to the caller, thus returning the new value after the atomic add. Note that it would be an error to instead have the return statement be `return *v;` this would introduce a subtle race condition, since `*v` would need to be reloaded from memory and could have a different value if another processor had executed an atomic add between the add and the load here.

AtomicInt32 1036

⟨*Do atomic add with gcc x86 inline assembly*⟩ ≡ 1036

```
int32_t origValue;
__asm__ __volatile__("lock\n"
                     "xaddl %0,%1"
                     : "=r"(origValue), "=m"(*v) : "0"(delta)
                     : "memory");
return origValue + delta;
```

We won't include other assembly language implementations of atomic primitives in the book text here.

Another useful atomic operation is "compare and swap." It takes a memory location and the value that the caller believes that the location stores (`oldValue`). If the memory location still holds that value when the atomic compare and swap executes, then the value `newValue` is stored; otherwise, memory is left unchanged. In either case, the current value stored at the location is returned.

⟨*Parallel Declarations*⟩ +≡

```
inline int32_t AtomicCompareAndSwap(AtomicInt32 *v, int32_t newValue,
                                    int32_t oldValue);
```

Compare and swap is a building block that can be used to build many other atomic operations. For example, the code below could be executed by multiple threads to compute the maximum of values computed by the threads:

```
int32_t oldval;
do {
    oldval = *ptr;
    newval = max(oldval, newval);
} while(AtomicCompareAndSwap(ptr, newval, oldval) != oldval);
```

If only a single thread is trying to update the memory location, the loop is successful the first time through; the value loaded into `oldval` is still the value at `*ptr` when `AtomicCompareAndSwap()` executes and so `newval` is successfully stored and `oldval` is returned. If multiple threads are executing concurrently, then another thread may update the value at `*ptr` between the thread's read into `oldval` and execution of `AtomicCompareAndSwap()`. In that case, the compare and swap fails, memory isn't updated, and another pass is taken through the loop to try again.

Another application of atomic compare and swap is lazy construction of data structures. Consider a tree data structure where each node has child node pointers, initially set to `NULL`. If code traversing the tree wants to create a new child at a node, code could be written like:

```
if (node->firstChild == NULL) {
    Type *newChild = new Type ...
    if (AtomicCompareAndSwap(&node->firstChild, newChild, NULL) != NULL)
        delete newChild;
}
/* node->firstChild != NULL now */
```

AtomicInt32 1036

The idea is that if the child is NULL, the thread speculatively creates and fully initializes the child node. Atomic compare and swap is then used to try to initialize the child pointer in the tree; if it's still NULL, then the new child is stored and made available to all threads. If the child pointer is no longer NULL, then another thread has initialized the child in the time between the current thread first seeing that it was NULL and the current thread trying to update it. In this case, the work done in the current thread turns out to have been wasted, but it can delete the locally created child node and continue execution, using the node created by the other thread.

This is a simple example of a *lock-free* algorithm. This approach has a few advantages compared to, for example, using a reader–writer mutex to manage updating the tree. First, there's no overhead of acquiring the reader mutex for regular tree traversal; doing so may in general require an expensive system call. Second, multiple threads can naturally concurrently update different parts of the tree. With a single reader–writer mutex, if one thread acquires the mutex to update one node in the tree, other threads won't be able to update other nodes. The "Further Reading" section at the end of the appendix has more pointers to lock-free algorithms.

Atomic add of floating-point values isn't directly available in most instruction sets, though it can be constructed with atomic compare and swap.

⟨*Parallel Declarations*⟩ +≡
```
inline float AtomicAdd(volatile float *val, float delta) {
    union bits { float f; int32_t i; };
    bits oldVal, newVal;
    do {
        oldVal.f = *val;
        newVal.f = oldVal.f + delta;
    } while (AtomicCompareAndSwap(((AtomicInt32 *)val),
                                  newVal.i, oldVal.i) != oldVal.i);
    return newVal.f;
}
```

A.9.3 MUTEXES

Instances of the Mutex class can't be allocated on the stack or directly as members of classes or structures but must be dynamically allocated. This allows us to ensure that they are allocated to start at the beginning of cache lines and that other objects are not allocated in their cache line, eliminating the risk of false sharing. The Create() and Destroy() methods here handle these operations.

⟨*Parallel Declarations*⟩ +≡
```
class Mutex {
public:
    static Mutex *Create();
    static void Destroy(Mutex *m);
private:
    ⟨Mutex Private Methods⟩
    ⟨System-dependent mutex implementation⟩
};
```

AtomicInt32 1036
Mutex 1038

The MutexLock structure manages acquisition and release of mutex locks; it waits in its constructor until the mutex is available, and when it goes out of scope and its destructor executes, the lock is released. Using a helper structure in this manner helps ensure that locks are released when no longer needed. If a mutex only needs to be held during the execution of some function, but the function has multiple return statements, this approach is more convenient than having to release the mutex before each return.

⟨*Parallel Declarations*⟩ +≡
```
struct MutexLock {
    MutexLock(Mutex &m);
    ~MutexLock();
private:
    Mutex &mutex;
};
```

RWMutex provides reader–writer mutexes, which allow multiple threads to acquire a lock for reading but only one thread at a time to acquire a lock for writing (and only when no other thread holds a reader lock).

⟨*Parallel Declarations*⟩ +≡
```
class RWMutex {
public:
    static RWMutex *Create();
    static void Destroy(RWMutex *m);
private:
    ⟨RWMutex Private Methods⟩
    ⟨System-dependent rw mutex implementation⟩
};
```

Similar to MutexLock, the RWMutexLock helper structure acquires and releases reader–writer mutexes. A parameter to the constructor indicates whether a read-only or a write lock is wanted. Furthermore, after a lock of one type has been acquired, methods allow requests to change to the other type. (These methods may of course need to block until other threads release their locks.)

⟨*Parallel Declarations*⟩ +≡
```
enum RWMutexLockType { READ, WRITE };
```

⟨*Parallel Declarations*⟩ +≡
```
struct RWMutexLock {
    RWMutexLock(RWMutex &m, RWMutexLockType t);
    ~RWMutexLock();
    void UpgradeToWrite();
    void DowngradeToRead();
private:
    RWMutexLockType type;
    RWMutex &mutex;
};
```

Mutex 1038
MutexLock 1039
RWMutex 1039
RWMutexLock 1039
RWMutexLockType 1039

A.9.4 TASK SYSTEM

`pbrt` is parallelized with a *task system* that executes independent *tasks* on the processing cores of the system. When a part of the system has work that can be done in parallel, it decomposes the work into some number of tasks and provides them to the task system for scheduling and execution. This style of parallelism has become widely and successfully used in game engines, for example. The 10.6 version of Mac OS® X provides a task system, *Grand Central Dispatch*, as a fundamental part of the system libraries, and the 2010 release of Microsoft® Visual Studio® provides a task system as part of the *Concurrency Runtime*. (The task system in this section is built on top of these system-level task systems if available.)

A strict definition of a task is that it is a computational job that is immediately ready to execute, independent of all other tasks, and able to run to completion. (In other words, a task won't go partway through its work and then ask the task scheduler to suspend it and resume it later, pending availability of some other resource.) This expectation implies that tasks aren't expected to hold mutexes for long periods of time. Given these constraints, the task system's job is relatively simple: it launches one hardware execution thread on each processor in the system where each thread runs code that removes a task from a task queue, executes it, and repeats, until no more tasks remain. Tasks can be launched in any order. Because persistent threads run on the processors (versus launching a new thread for each unit of work), tasks can be fine-grained—they don't need to perform large amounts of computation to amortize the operating system's thread launching overhead.

This style of expressing parallelism in software systems often leads to clean code implementations and good parallel scalability. The requirements (like independence from other tasks and ability to run to completion) lead to high performance when the tasks execute, and the code that implements the work that tasks do tends to have few parallel synchronization calls in it; most of this is isolated in the task system itself. The programmer's main task is to decompose the computation into as many fine-grained tasks as possible and then not worry about the details of how they run. Furthermore, because the programmer is encouraged to create many fine-grained tasks (rather than decompose the work into a number of units equal to the number of processing cores present), the system can load-balance well when tasks have variable amounts of computation and scale well as more cores are available in the future.

`TasksInit()` is called at system startup time. It launches a task worker thread for each of the cores in the system (or, it launches the number of threads manually specified by the user), where each thread calls a function that waits for work on the task queue and takes and runs jobs as they become available. Thus, we have the main thread of execution continuing to run and independent of the task execution threads. In `pbrt`, usually either the task worker threads will be active running tasks and the main thread will be waiting for the tasks to all finish before continuing, or the main thread will be running while the worker threads are waiting for new tasks to be created. In other task-based software systems, it's often the case that the main thread and some tasks execute concurrently. This simpler approach makes `pbrt`'s parallelization easier to understand, at a small cost to performance.

⟨*Parallel Declarations*⟩ +≡
```
void TasksInit();
```

`TasksCleanup()` cleans up the resources allocated for the threads. It is an error (detected by an `Assert()` test) to call this function while tasks are still running.

⟨*Parallel Declarations*⟩ +≡
```
void TasksCleanup();
```

Tasks are implemented by inheriting from the abstract `Task` base-class. Implementations of this class gather the data needed to execute the task and implement the `Run()` method. See the `SamplerRendererTask` implementation in Section 1.3.4 as an example.

⟨*Parallel Declarations*⟩ +≡
```
class Task {
public:
    virtual ~Task();
    virtual void Run() = 0;
};
```

The `EnqueueTasks()` function adds the given tasks to the work queue and returns immediately. In `pbrt`, the main thread generally will want to wait for all of the tasks to finish before continuing execution; the `WaitForAllTasks()` function does just this. Alternatively, if the results from the tasks weren't needed immediately, the main thread could continue execution and call `WaitForAllTasks()` later.

The implementation of `pbrt`'s task system is relatively simple: there's a single task queue, protected by a mutex, shared by all task worker threads. More sophisticated task system implementations have multiple task queues, each with different priorities, and per-thread local task queues, allowing worker threads to quickly retrieve the next task to run without contention at a centralized task queue. In this approach, when a thread's local task queue is empty, it steals work from another thread's queue.

More sophisticated task systems also support finer-grained control to wait for task completion, making it possible to mark a set of tasks as a group and wait for their completion only, while other tasks continue to run. `pbrt` doesn't currently need this type of functionality, so it's not currently available in the task system.

⟨*Parallel Declarations*⟩ +≡
```
void EnqueueTasks(const vector<Task *> &tasks);
void WaitForAllTasks();
```

The last function in this section is `NumSystemCores()`, which returns the number of processing cores in the system. Its implementation is highly operating system dependent.

SamplerRendererTask 29
Task 1041
WaitForAllTasks() 1041

⟨*Parallel Declarations*⟩ +≡
```
int NumSystemCores();
```

FURTHER READING

Hacker's Delight (Warren 2006), is a delightful and thought-provoking exploration of the bit-twiddling algorithms like those used in some of the utility routines in this appendix. Sean Anderson has a Web page filled with a collection of bit-twiddling techniques like the ones in the `IsPowerOf2()` and `RoundUpPow2()` functions at *graphics.stanford.edu/~seander/bithacks.html*. See Kotay's Web page at *www.stereopsis.com/sree/fpu2006.html* for an extensive discussion of efficient floating-point to integer conversions.

Numerical Recipes, by Press et al. (1992), and Atkinson's book (1993) on numerical analysis both discuss algorithms for matrix inversion and solving linear systems.

Detailed information about the random number generator used in `pbrt`, including the original paper that describes its derivation, is available at *www.math.keio.ac.jp/~matumoto/emt.html*.

Many papers have been written on cache-friendly programming techniques; only a few are surveyed here. Ericson's chapter on high-performance programming techniques has very good coverage of this topic (Ericson 2004). Lam, Rothberg, and Wolf (1991) investigated blocking (tiling) for improving cache performance and developed techniques for selecting appropriate block sizes, given the size of the arrays and the cache size. Grunwald, Zorn, and Henderson (1993) were one of the first groups of researchers to investigate the interplay between memory allocation algorithms and the cache behavior of applications.

In `pbrt`, we only worry about cache layout issues for dynamically allocated data. However, Calder et al. (1998) show a profile-driven system that optimizes memory layout of global variables, constant values, data on the stack, and dynamically allocated data from the heap in order to reduce cache conflicts among them all, giving an average 30% reduction in data cache misses for the applications they studied.

Blocking for tree data structures was investigated by Chilimbi et al. (1999b); they ensured that nodes of the tree and a few levels of its children were allocated contiguously. Among other applications, they applied their tool to the layout of the acceleration octree in the *Radiance* renderer and reported a 42% speedup in run time. Chilimbi et al. (1999a) also evaluated the effectiveness of reordering fields inside structures to improve locality.

Drepper's paper (2007) is a useful resource for understanding performance issues related to caches, cache coherence, and main memory access, particularly in multicore systems.

Samet's book on octrees is the canonical reference on that data structure (Samet 1990), and de Berg et al.'s book on computational geometry has extensive information about kd-trees (de Berg et al. 2000).

Given the computational cost of ray tracing, there has been interest in parallel algorithms for ray tracing since shortly after the algorithm was first introduced (Cleary et al. 1983; Green and Paddon 1989; Badouel and Priol 1989). With the computational capabilities available in modern CPUs, researchers have started to demonstrate interactive ray tracing using tens of processors. For example, Parker et al. (1999) developed an interactive ray tracer on a shared-memory computer with 64 processors, and Wald and collaborators ray-traced complex scenes at interactive rates on a cluster of PCs (Wald et al. 2001a, 2002; Wald, Slusallek, and Benthin 2001; Wald, Benthin, and Slusallek 2003).

`IsPowerOf2()` 1001
`RoundUpPow2()` 1002

Task systems like the one in pbrt have been widely used in recent years. Blumofe et al. (1996) described the task scheduler in Cilk, and Blumofe and Leiserson (1999) describe the work-stealing algorithm that is the mainstay of current high-performance task systems. Boehm's paper *Threads Cannot Be Implemented as a Library* (2005) makes the remarkable (and disconcerting) observation that multithreading cannot be reliably implemented without the compiler having expicit knowledge of the fact that multithreaded execution is expected. Boehm presented a number of examples that demonstrate the danger in current compilers and language standards like C and C++. Fortunately, the forthcoming C++0x standard addresses the issues he has identified.

EXERCISES

❶ A.1 Modify MemoryArena so that it just calls new for each memory allocation. Render images of a few scenes and measure how much slower pbrt runs. Can you quantify how much of this is due to different cache behavior and how much is due to overhead in the dynamic memory management routines?

❶ A.2 Change the BlockedArray class so that it doesn't do any blocking and just uses a linear addressing scheme for the array. Measure the change in pbrt's performance as a result. (Scenes with many image map textures are most likely to show any differences, since the MIPMap class is a key user of BlockedArray.)

❷ A.3 The KdNode for the KdTree can be brought down to use just four bytes of storage; making this change may further improve its memory performance. Modify the KdNode to just store the split position and split axis in four bytes, using low two bits of the floating-point mantissa to store the split position. Then, modify the tree construction routine to build a *left-balanced* kd-tree, where the tree's topology is organized such that for the node at position i in the array of nodes, the left child is at 2*i, the right child is at 2*i+1, and the tree is balanced such that only the first nNodes elements of the array are used. How does this change affect performance for scenes that use photon mapping? Discuss why this change makes the performance difference that it did.

BlockedArray 1017
KdTree 1029
MemoryArena 1015
MIPMap 530

This page intentionally left blank



SCENE DESCRIPTION INTERFACE

This appendix describes the application programming interface (API) that is used to describe the scene to be rendered to `pbrt`. Users of the renderer typically don't call the functions in this interface directly, but instead describe their scenes using the text file format described in the file `docs/fileformat.pdf` in the `pbrt` distribution. The statements in these text files have a direct correspondence to the API functions described here.

The need for such an interface to the renderer is clear: there must be a convenient way in which all of the properties of the scene to be rendered can be communicated to the renderer. The interface should be well defined and general purpose, so that future extensions to the system fit into its structure cleanly. It shouldn't be too complicated, so that it's easy to describe scenes, but it should be expressive enough that it doesn't leave any of the renderer's capabilities hidden.

A key decision to make when designing a rendering API is whether to expose the system's internal algorithms and structures or offer a high-level abstraction for describing the scene. These have historically been the two main approaches to scene description in graphics: the interface may specify *how* to render the scene, configuring a rendering pipeline at a low level using deep knowledge of the renderer's internal algorithms, or it may specify *what* the scene's objects, lights, and material properties are and leave it to the renderer to decide how to transform that description into the best possible image.

The first approach has been successfully used for interactive graphics. In APIs such as OpenGL® or Direct3D®, it is not possible to just mark an object as a mirror and have reflections appear automatically; rather, the user must choose an algorithm for rendering reflections, render the scene multiple times (e.g., to generate an environment map), store those images in a texture, and then configure the graphics pipeline to use the environment map when rendering the reflective object. The advantage of this approach

is that the full flexibility of the rendering pipeline is exposed to the user, making it possible to carefully control the actual computation being done and to use the pipeline very efficiently. Furthermore, because APIs like these impose a very thin abstraction layer between the user and the renderer, the user can be confident that unexpected inefficiencies won't be introduced by the API.

The second approach to scene description, based on describing the geometry, materials, and lights at a higher level of abstraction, has been most successful for applications like high-quality offline rendering. There, users are generally willing to cede control of the low-level rendering details to the renderer in exchange for the ability to specify the scene's properties at a high level. An important advantage of the high-level approach is that the implementations of these renderers have greater freedom to make major changes to the internal algorithms of the system, since the API exposes less of them.

For `pbrt`, we will use an interface based on the descriptive approach. Because `pbrt` is fundamentally physically based, the API is necessarily less flexible in some ways than APIs for many nonphysically based rendering packages. For example, it is not possible to have some lights illuminate only some objects in the scene.

Another key decision to make in graphics API design is whether to use an immediate mode or a retained mode style. In an immediate mode API, the user specifies the scene via a stream of commands that the renderer processes as they arrive. In general, the user cannot make changes to the scene description data already specified (e.g., "change the material of that sphere I described previously from plastic to glass"); once it has been given to the renderer, the information is no longer accessible to the user. Retained mode APIs give the user some degree of access to the data structures that the renderer has built to represent the scene. The user can then modify the scene description in a variety of ways before finally instructing the renderer to render the scene.

Immediate mode has been very successful for interactive graphics APIs since it allows graphics hardware to draw the objects in the scene as they are supplied by the user. Since they do not need to build data structures to store the scene and since they can apply techniques like immediately culling objects that are outside of the viewing frustum without worrying that the user will change the camera position before rendering, these APIs have been key to high-performance interactive graphics.

For ray-tracing-based renderers like `pbrt`, where the entire scene must be described and stored in memory before rendering can begin, some of these advantages of an immediate mode interface aren't applicable. Nonetheless, we will use immediate mode semantics in our API, since it leads to a clean and straightforward scene description language. This choice makes it more difficult to use `pbrt` for applications like quickly rerendering a scene after making a small change to it (e.g., by moving a light source) and may make rendering animations less straightforward, since the entire scene needs to be redescribed for each frame of an animation. Adding a retained mode interface to `pbrt` would be a challenging but useful project.

`pbrt`'s rendering API consists of just over 40 carefully chosen functions, all of which are declared in the `core/api.h` header file. The implementation of these functions is in `core/api.cpp`. This appendix will focus on the general process of turning the API function calls into instances of the classes that represent the scenes.

B.1 PARAMETER SETS

A key problem that a rendering API must address is extensibility: if a developer has created new implementations of shapes, cameras, or the like for pbrt, it shouldn't be necessary to modify the API. In particular, the API should be as unaware as possible of what particular parameters these objects take and what their semantics are. pbrt uses the ParamSet class to address this issue. This class handles collections of named parameters and their values in a generic way. For example, it might record that there is a single floating-point value named "radius" with a value of 2.5, and an array of four color values named "specular" with various color values. The ParamSet provides methods for both setting and retrieving values from these kinds of generic parameter lists. It is defined in core/paramset.h and core/paramset.cpp.

Most of pbrt's API routines take a ParamSet as one of their parameters; for example, the shape creation routine, pbrtShape(), takes a string giving the name of the shape to make and a ParamSet with parameters for it. The creation routine of the corresponding shape implementation is called with the ParamSet passed along as a parameter. This style makes the API's implementation very straightforward.

⟨*ParamSet Declarations*⟩ ≡
```
class ParamSet {
public:
    ⟨ParamSet Public Methods 1049⟩
private:
    ⟨ParamSet Private Data 1047⟩
};
```

A ParamSet can hold nine types of parameters: Booleans, integers, floating-point values, points, vectors, normals, spectra, strings, and the names of Textures that are being used as parameters for Materials and other Textures. Internally, it stores a vector of named values for each of the different types that it stores; each parameter is represented by a ParamSetItem of the appropriate type. This representation means that searching for a given parameter takes $O(n)$ time, where n is the number of parameters of the parameter's type. In practice, there are just a handful of parameters to any function, so a more time-efficient representation isn't necessary.

Material 483
Normal 65
ParamSet 1047
ParamSetItem 1048
pbrtShape() 1065
Point 63
Reference 1011
Spectrum 263
Texture 519
Vector 57

⟨*ParamSet Private Data*⟩ ≡ 1047
```
vector<Reference<ParamSetItem<bool> > > bools;
vector<Reference<ParamSetItem<int> > > ints;
vector<Reference<ParamSetItem<float> > > floats;
vector<Reference<ParamSetItem<Point> > > points;
vector<Reference<ParamSetItem<Vector> > > vectors;
vector<Reference<ParamSetItem<Normal> > > normals;
vector<Reference<ParamSetItem<Spectrum> > > spectra;
vector<Reference<ParamSetItem<string> > > strings;
vector<Reference<ParamSetItem<string> > > textures;
```

B.1.1 THE ParamSetItem STRUCTURE

The ParamSetItem structure stores all of the relevant information about a single parameter, such as its name, its base type, and its value(s). For example (using the syntax from pbrt's input files), the foo parameter

```
"float foo" [ 0 1 2 3 4 5 ]
```

has a base type of float, and six values have been supplied for it. It would be represented by a single ParamSetItem<float>. To simplify determining when the data a ParmSetItem stores can be freed, this structure is reference counted.

⟨*ParamSet Declarations*⟩ +≡
```
template <typename T> struct ParamSetItem : public ReferenceCounted {
    ⟨ParamSetItem Public Methods⟩
    ⟨ParamSetItem Data 1048⟩
};
```

The ParamSetItem directly initializes its members from the arguments and makes a copy of the values.

⟨*ParamSetItem Methods*⟩ ≡
```
template <typename T>
ParamSetItem<T>::ParamSetItem(const string &n, const T *v, int ni) {
    name = n;
    nItems = ni;
    data = new T[nItems];
    for (int i = 0; i < nItems; ++i) data[i] = v[i];
    lookedUp = false;
}
```

The Boolean value lookedUp is set to true after the value has been retrieved from the ParamSet. This makes it possible to print warning messages if any parameters were added to the parameter set but never used, which typically indicates a misspelling in the scene description file or other user error.

⟨*ParamSetItem Data*⟩ ≡ 1048
```
string name;
int nItems;
T *data;
mutable bool lookedUp;
```

ParamSetItem 1048
ParamSetItem::data 1048
ParamSetItem::lookedUp 1048
ParamSetItem::name 1048
ParamSetItem::nItems 1048
ReferenceCounted 1010

B.1.2 ADDING TO THE PARAMETER SET

To add an entry to the parameter set, the user calls the appropriate ParamSet method, passing the name of the parameter, a pointer to its data, and the number of data items. These methods first remove previous values for this parameter, if any.

⟨*ParamSet Methods*⟩ ≡
```
void ParamSet::AddFloat(const string &name, const float *data,
                        int nItems) {
    EraseFloat(name);
    floats.push_back(new ParamSetItem<float>(name, data, nItems));
}
```

We won't include the rest of the methods to add data to the `ParamSet`, but we do include their prototypes here for reference. The erasure methods are also straightforward and won't be included here.

⟨*ParamSet Public Methods*⟩ ≡ 1047
```
void AddInt(const string &, const int *, int nItems);
void AddBool(const string &, const bool *, int nItems);
void AddPoint(const string &, const Point *, int nItems);
void AddVector(const string &, const Vector *, int nItems);
void AddNormal(const string &, const Normal *, int nItems);
void AddString(const string &, const string *, int nItems);
void AddTexture(const string &, const string &);
```

A number of different methods for adding spectral data are provided, making it easy for the data to be supplied with a variety of representations.

⟨*ParamSet Public Methods*⟩ +≡ 1047
```
void AddRGBSpectrum(const string &, const float *, int nItems);
void AddXYZSpectrum(const string &, const float *, int nItems);
void AddBlackbodySpectrum(const string &, const float *, int nItems);
void AddSampledSpectrumFiles(const string &, const char **, int nItems);
void AddSampledSpectrum(const string &, const float *, int nItems);
```

B.1.3 LOOKING UP VALUES IN THE PARAMETER SET

To retrieve a parameter value from a set, it is necessary to loop through the entries of the requested type and return the appropriate value, if any. There are two versions of the lookup method for each parameter type: a simple one for parameters that have a single data value, and a more general one that returns a pointer to the possibly multiple values of more complex types. The first method mostly serves to reduce the amount of code needed in routines that retrieve parameter values.

The methods that look up a single item (e.g., `FindOneFloat()`) take the name of the parameter and a default value. If the parameter is not found, the default value is returned. This makes it easy to write initialization code like

```
float radius = params.FindOneFloat("radius", 1.f);
```

In this case, it is not an error if there isn't a "radius" parameter; the default value will be used instead. If the caller wants to detect a missing parameter and issue an error, the second variant of lookup method should be used, since they return a `NULL` pointer if the parameter isn't found.

Normal 65
ParamSet 1047
ParamSetItem 1048
Point 63
Vector 57

⟨*ParamSet Methods*⟩ +≡
```
float ParamSet::FindOneFloat(const string &name, float d) const {
    for (uint32_t i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name && floats[i]->nItems == 1) {
            floats[i]->lookedUp = true;
            return *(floats[i]->data);
        }
    return d;
}
```

As earlier, here are the declarations of the analogous methods for the remaining types:

⟨*ParamSet Public Methods*⟩ +≡ 1047
```
int FindOneInt(const string &, int d) const;
bool FindOneBool(const string &, bool d) const;
Point FindOnePoint(const string &, const Point &d) const;
Vector FindOneVector(const string &, const Vector &d) const;
Normal FindOneNormal(const string &, const Normal &d) const;
Spectrum FindOneSpectrum(const string &,
                         const Spectrum &d) const;
string FindOneString(const string &, const string &d) const;
string FindTexture(const string &) const;
```

The second kind of lookup method returns a pointer to the data if the data is present and returns the number of data items in `nItems`.

⟨*ParamSet Methods*⟩ +≡
```
const float *ParamSet::FindFloat(const string &name, int *n) const {
    for (uint32_t i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name) {
            *n = floats[i]->nItems;
            floats[i]->lookedUp = true;
            return floats[i]->data;
        }
    return NULL;
}
```

These are the prototypes for the rest of the lookup functions for the other types:

⟨*ParamSet Public Methods*⟩ +≡ 1047
```
const int *FindInt(const string &, int *nItems) const;
const bool *FindBool(const string &, int *nItems) const;
const Point *FindPoint(const string &, int *nItems) const;
const Vector *FindVector(const string &, int *nItems) const;
const Normal *FindNormal(const string &, int *nItems) const;
const Spectrum *FindSpectrum(const string &, int *nItems) const;
const string *FindString(const string &, int *nItems) const;
```

Because the user may misspell parameter names in the scene description file, the `ParamSet` also provides a `ReportUnused()` function, not included here, that goes through the pa-

Normal 65
ParamSet 1047
ParamSet::floats 1047
ParamSetItem::data 1048
ParamSetItem::lookedUp 1048
ParamSetItem::name 1048
ParamSetItem::nItems 1048
Point 63
Spectrum 263
Vector 57

rameter set and reports if any of the parameters present were never looked up, checking the `ParamSetItem::lookedUp` member variable. For any items where this variable stores the value `false`, it is likely that the user has given an incorrect parameter.

⟨*ParamSet Public Methods*⟩ +≡ 1047
```
void ReportUnused() const;
```

The `ParamSet::Clear()` method clears all of the individual parameter vectors. The corresponding `ParamSetItems` will in turn be freed if their reference count goes to zero.

⟨*ParamSet Public Methods*⟩ +≡ 1047
```
void Clear();
```

B.2 INITIALIZATION AND RENDERING OPTIONS

We now have the machinery to describe the routines that make up the rendering API. Before any other API functions can be called, the rendering system must be initialized by a call to `pbrtInit()`. Similarly, when rendering is done, `pbrtCleanup()` should be called; this handles final cleanup of the system. The definitions of these two functions will be filled in at a number of points throughout the rest of this appendix.

A few systemwide options are passed to `pbrtInit()` using the `Options` structure here.

⟨*Global Forward Declarations*⟩ +≡
```
struct Options {
    Options() { nCores = 0;
                quickRender = quiet = openWindow = verbose = false;
                imageFile = ""; }
    int nCores;
    bool quickRender;
    bool quiet, verbose;
    bool openWindow;
    string imageFile;
};
```

⟨*API Function Definitions*⟩ ≡
```
void pbrtInit(const Options &opt) {
    PbrtOptions = opt;
    ⟨API Initialization 1053⟩
    SampledSpectrum::Init();
}
```

Options 1051
ParamSet::Clear() 1051
ParamSetItem::lookedUp 1048
PbrtOptions 1051
SampledSpectrum::Init() 271

The options are stored in a global variable for easy access by other parts of the system. This variable is only used in a read-only fashion by the system after initialization in `pbrtInit()`.

⟨*API Global Variables*⟩ ≡
```
Options PbrtOptions;
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtCleanup() {
    ProbesCleanup();
    ⟨API Cleanup 1053⟩
}
```

After the system has been initialized, a subset of the API routines is available. Legal calls at this point are those that set general rendering options like the camera and sampler properties, the type of film to be used, and so on, but the user is not yet allowed to start to describe the lights, shapes, and materials in the scene.

After the overall rendering options have been set, the `pbrtWorldBegin()` function locks them in; it is no longer legal to call the routines that set them. At this point, the user can begin to describe the geometric primitives and lights that are in the scene. This separation of global versus scene-specific information can help simplify the implementation of the renderer. For example, consider a spline patch shape that tessellates itself into triangles. This shape might compute the required size of its generated triangles based on the area of the screen that it covers. If the camera's position and image resolution are guaranteed not to change after the shape is created, then the shape can potentially do the tessellation work immediately at creation time.

Once the scene has been fully specified, the `pbrtWorldEnd()` routine is called. At this point, the renderer knows that the scene description is complete and that rendering can begin. The image will be rendered and written to a file before `pbrtWorldEnd()` returns. The user may then specify new options for another frame of an animation, and then another `pbrtWorldBegin()`/`pbrtWorldEnd()` block to describe the geometry for the next frame, repeating as many times as desired. The remainder of this section will discuss the routines related to setting rendering options. Section B.3 describes the routines for specifying the scene inside the world block.

B.2.1 STATE TRACKING

There are three distinct states that the renderer's API can be in:

- *Uninitialized:* Before `pbrtInit()` has been called, no other API calls are legal.
- *Option block:* Outside a `pbrtWorldBegin()` and `pbrtWorldEnd()` pair, scenewide global options may be set.
- *World block:* Inside a `pbrtWorldBegin()` and `pbrtWorldEnd()` pair, the scene may be described.

A module static variable `currentApiState` starts out with value `STATE_UNINITIALIZED`, indicating that the API system hasn't yet been initialized. Its value is updated appropriately by `pbrtInit()`, `pbrtWorldBegin()`, and `pbrtCleanup()`.

⟨*API Static Data*⟩ ≡
```
#define STATE_UNINITIALIZED  0
#define STATE_OPTIONS_BLOCK  1
#define STATE_WORLD_BLOCK    2
static int currentApiState = STATE_UNINITIALIZED;
```

pbrtInit() 1051
pbrtWorldBegin() 1059
pbrtWorldEnd() 1071
STATE_UNINITIALIZED 1052

Now we can start to define the implementation of pbrtInit(). pbrtInit() first makes sure that it hasn't already been called and then sets the currentApiState variable to STATE_OPTIONS_BLOCK to indicate that the scenewide options can be specified.

⟨*API Initialization*⟩ ≡ 1051
```
if (currentApiState != STATE_UNINITIALIZED)
    Error("pbrtInit() has already been called.");
currentApiState = STATE_OPTIONS_BLOCK;
```

Similarly, pbrtCleanup() makes sure that pbrtInit() has been called and that we're not in the middle of a pbrtWorldBegin()/pbrtWorldEnd() block before resetting the state to the uninitialized state.

⟨*API Cleanup*⟩ ≡ 1052
```
if (currentApiState == STATE_UNINITIALIZED)
    Error("pbrtCleanup() called without pbrtInit().");
else if (currentApiState == STATE_WORLD_BLOCK)
    Error("pbrtCleanup() called while inside world block.");
currentApiState = STATE_UNINITIALIZED;
```

All API procedures that are only valid in particular states invoke a state verification macro like VERIFY_INITIALIZED(), to make sure that currentApiState holds an appropriate value. If the states don't match, an error message is printed and the function immediately returns.

⟨*API Macros*⟩ ≡
```
#define VERIFY_INITIALIZED(func) \
if (currentApiState == STATE_UNINITIALIZED) { \
    Error("pbrtInit() must be before calling \"%s()\". " \
          "Ignoring.", func); \
    return; \
} else /* swallow trailing semicolon */
```

The implementations of VERIFY_OPTIONS() and VERIFY_WORLD() are analogous.

B.2.2 TRANSFORMATIONS

As the scene is being described, pbrt maintains *current transformation matrices* (CTMs), one for each of a number of points in time. If the transformations are different, then they describe an animated transformation. (Recall, for example, that the AnimatedTransform class defined in Section 2.9.3 stores two transformation matrices for two given times.) A number of API calls are available to modify the CTMs; when objects like shapes, cameras, and lights are created, the CTMs are passed to their constructor to define the transformation from their local coordinate system to world space.

activeTransformBits 1054
AnimatedTransform 96
currentApiState 1052
curTransform 1054
Error() 1005
pbrtInit() 1051
STATE_OPTIONS_BLOCK 1052
STATE_UNINITIALIZED 1052
STATE_WORLD_BLOCK 1052
TransformSet 1054

The code below stores two CTMs in the module-local curTransform variable. They are represented by the TransformSet class, to be defined shortly, which stores a fixed number of transformations. The activeTransformBits variable is a bit-vector indicating which of the CTMs are active; the active transforms are updated when the transformation-related API calls are made, while the others are unchanged. This mechanism allows the user to modify some of the CTMs selectively in order to define animated transformations.

⟨*API Static Data*⟩ +≡
```
static TransformSet curTransform;
static int activeTransformBits = ALL_TRANSFORMS_BITS;
```

The implementation here just stores two transformation matrices, one that defines the CTM for the starting time (provided via the `pbrtTransformTimes()` call, defined in a few pages), and the other for the ending time.

⟨*API Local Classes*⟩ ≡
```
#define MAX_TRANSFORMS 2
#define START_TRANSFORM_BITS (1 << 0)
#define END_TRANSFORM_BITS   (1 << 1)
#define ALL_TRANSFORMS_BITS  ((1 << MAX_TRANSFORMS) - 1)
```

`TransformSet` is a small utility class that stores an array of transformations and provides some utility routines for managing them.

⟨*API Local Classes*⟩ +≡
```
struct TransformSet {
    ⟨TransformSet Public Methods 1054⟩
private:
    Transform t[MAX_TRANSFORMS];
};
```

An accessor function is provided to access the individual `Transforms`.

⟨*TransformSet Public Methods*⟩ ≡ 1054
```
Transform &operator[](int i) {
    return t[i];
}
```

The `Inverse()` method returns a new `TransformSet` that holds the inverses of the individual `Transforms`.

⟨*TransformSet Public Methods*⟩ +≡ 1054
```
friend TransformSet Inverse(const TransformSet &ts) {
    TransformSet t2;
    for (int i = 0; i < MAX_TRANSFORMS; ++i)
        t2.t[i] = Inverse(ts.t[i]);
    return t2;
}
```

The actual transformation functions are straightforward. Because the CTM is used both for the rendering options and the scene description phases, these routines only need to verify that `pbrtInit()` has been called.

⟨*API Function Definitions*⟩ +≡
```
void pbrtIdentity() {
    VERIFY_INITIALIZED("Identity");
    FOR_ACTIVE_TRANSFORMS(curTransform[i] = Transform();)
}
```

ALL_TRANSFORMS_BITS 1054
curTransform 1054
FOR_ACTIVE_TRANSFORMS() 1055
Inverse() 1021
MAX_TRANSFORMS 1054
pbrtInit() 1051
pbrtTransformTimes() 1057
Transform 76
TransformSet 1054
VERIFY_INITIALIZED 1053

The FOR_ACTIVE_TRANSFORMS() macro encapsulates the logic for determining which of the CTMs is active. The given expression is evaluated only for the active transforms.

⟨*API Macros*⟩ +≡
```
#define FOR_ACTIVE_TRANSFORMS(expr) \
    for (int i = 0; i < MAX_TRANSFORMS; ++i) \
        if (activeTransformBits & (1 << i)) { expr }
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtTranslate(float dx, float dy, float dz) {
    VERIFY_INITIALIZED("Translate");
    FOR_ACTIVE_TRANSFORMS(curTransform[i] =
        curTransform[i] * Translate(Vector(dx, dy, dz));)
}
```

Most of the rest of the functions are similarly defined, so we will not show their definitions here. pbrt also provides pbrtConcatTransform() and pbrtTransform() functions to allow the user to specify an arbitrary matrix to postmultiply or replace the active CTM(s), respectively.

⟨*API Function Declarations*⟩ ≡
```
void pbrtRotate(float angle, float ax, float ay, float az);
void pbrtScale(float sx, float sy, float sz);
void pbrtLookAt(float ex, float ey, float ez,
                float lx, float ly, float lz,
                float ux, float uy, float uz);
void pbrtConcatTransform(float transform[16]);
void pbrtTransform(float transform[16]);
```

It can be useful to make a named copy of the CTM so that it can be referred to later. For example, to place a light at the camera's position, it is useful to first apply the transformation into the camera coordinate system, since then the light can just be placed at the origin (0, 0, 0). This way, if the camera position is changed and the scene is rerendered, the light will move with it. The pbrtCoordinateSystem() function copies the current TransformSet into the namedCoordinateSystems associative array, and pbrtCoordSysTransform() loads a named set of CTMs.

⟨*API Static Data*⟩ +≡
```
static map<string, TransformSet> namedCoordinateSystems;
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtCoordinateSystem(const string &name) {
    VERIFY_INITIALIZED("CoordinateSystem");
    namedCoordinateSystems[name] = curTransform;
}
```

activeTransformBits 1054
curTransform 1054
FOR_ACTIVE_TRANSFORMS() 1055
MAX_TRANSFORMS 1054
namedCoordinateSystems 1055
pbrtConcatTransform() 1055
pbrtCoordinateSystem() 1055
pbrtCoordSysTransform() 1056
pbrtTransform() 1055
TransformSet 1054
Vector 57
VERIFY_INITIALIZED 1053

⟨*API Function Definitions*⟩ +≡
```
void pbrtCoordSysTransform(const string &name) {
    VERIFY_INITIALIZED("CoordSysTransform");
    if (namedCoordinateSystems.find(name) !=
        namedCoordinateSystems.end())
        curTransform = namedCoordinateSystems[name];
    else
        Warning("Couldn't find named coordinate system \"%s\"",
                name.c_str());
}
```

B.2.3 OPTIONS

All of the rendering options that are set before the `pbrtWorldBegin()` call are stored in a `RenderOptions` structure. This structure contains public data members that are set by API calls and methods that help create objects used by the rest of `pbrt` for rendering.

⟨*API Local Classes*⟩ +≡
```
struct RenderOptions {
    ⟨RenderOptions Public Methods 1072⟩
    ⟨RenderOptions Public Data 1057⟩
};
```

⟨*API Local Classes*⟩ +≡
```
RenderOptions::RenderOptions() {
    ⟨RenderOptions Constructor Implementation 1057⟩
}
```

A single `static` instance of a `RenderOptions` structure is available to the rest of the API functions:

⟨*API Static Data*⟩ +≡
```
static RenderOptions *renderOptions = NULL;
```

When `pbrtInit()` is called, it allocates a `RenderOptions` structure that initially holds default values for all of its options:

⟨*API Initialization*⟩ +≡ 1051
```
renderOptions = new RenderOptions;
```

The `renderOptions` variable is freed by `pbrtCleanup()`:

⟨*API Cleanup*⟩ +≡ 1052
```
delete renderOptions;
renderOptions = NULL;
```

A few calls are available to indicate which of the CTMs are active.

curTransform 1054
namedCoordinateSystems 1055
pbrtInit() 1051
pbrtWorldBegin() 1059
RenderOptions 1056
renderOptions 1056
VERIFY_INITIALIZED 1053
Warning() 1005

⟨*API Function Definitions*⟩ +≡
```
void pbrtActiveTransformAll() {
    activeTransformBits = ALL_TRANSFORMS_BITS;
}
void pbrtActiveTransformEndTime() {
    activeTransformBits = END_TRANSFORM_BITS;
}
void pbrtActiveTransformStartTime() {
    activeTransformBits = START_TRANSFORM_BITS;
}
```

The two times at which the two CTMs are defined are provided via `pbrtTransformTimes()`. By default, the start time is zero and the end time is one.

⟨*API Function Definitions*⟩ +≡
```
void pbrtTransformTimes(float start, float end) {
    VERIFY_OPTIONS("TransformTimes");
    renderOptions->transformStartTime = start;
    renderOptions->transformEndTime = end;
}
```

⟨*RenderOptions Public Data*⟩ ≡ 1056
```
float transformStartTime, transformEndTime;
```

The API functions for setting the rest of the rendering options are mostly similar in both their interface and their implementation. For example, `pbrtPixelFilter()` specifies the kind of `Filter` to be used for filtering image samples. It takes two parameters: a string giving the name of the filter to use and a `ParamSet` giving the parameters to the filter. When this function is called, all that needs to be done is to verify that the API is in an appropriate state for `pbrtPixelFilter()` to be called and store the name of the filter and its parameters in `renderOptions`.

⟨*API Function Definitions*⟩ +≡
```
void pbrtPixelFilter(const string &name, const ParamSet &params) {
    VERIFY_OPTIONS("PixelFilter");
    renderOptions->FilterName = name;
    renderOptions->FilterParams = params;
}
```

activeTransformBits 1054
ALL_TRANSFORMS_BITS 1054
END_TRANSFORM_BITS 1054
Filter 393
ParamSet 1047
pbrtPixelFilter() 1057
pbrtTransformTimes() 1057
renderOptions 1056
RenderOptions::
FilterName 1057
RenderOptions::
FilterParams 1057
START_TRANSFORM_BITS 1054

⟨*RenderOptions Public Data*⟩ +≡ 1056
```
string FilterName;
ParamSet FilterParams;
```

The default filter function is set to the Box filter. If no specific filter is specified in the scene description file, then, since the default `ParamSet` has no key-value pairs, the filter will be created based on its default parameter settings.

⟨*RenderOptions Constructor Implementation*⟩ ≡ 1056
```
FilterName = "box";
```

Most of the rest of the rendering option setting API calls are similar; they simply store their arguments in `renderOptions`. Therefore, we will only include the declarations of these functions here. The options controlled by each function should be apparent from its name; more information about the legal parameters to each of these routines can be found in the `docs/fileformat.pdf` file.

⟨*API Function Declarations*⟩ +≡
```
void pbrtFilm(const string &type, const ParamSet &params);
void pbrtSampler(const string &name, const ParamSet &params);
void pbrtAccelerator(const string &name, const ParamSet &params);
void pbrtSurfaceIntegrator(const string &name, const ParamSet &params);
void pbrtVolumeIntegrator(const string &name, const ParamSet &params);
void pbrtRenderer(const string &name, const ParamSet &params);
```

`pbrtCamera()` is slightly different than the other options, since the world-to-camera transformation needs to be recorded. The CTM is used by `pbrtCamera()` to initialize this value, and the camera coordinate system transformation is also stored for possible future use by `pbrtCoordSysTransform()`.

⟨*RenderOptions Public Data*⟩ +≡ 1056
```
string CameraName;
ParamSet CameraParams;
TransformSet CameraToWorld;
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtCamera(const string &name, const ParamSet &params) {
    VERIFY_OPTIONS("Camera");
    renderOptions->CameraName = name;
    renderOptions->CameraParams = params;
    renderOptions->CameraToWorld = Inverse(curTransform);
    namedCoordinateSystems["camera"] = renderOptions->CameraToWorld;
}
```

The default camera is set to use a perspective projection:

⟨*RenderOptions Constructor Implementation*⟩ +≡ 1056
```
CameraName = "perspective";
```

B.3 SCENE DEFINITION

After the user has set up the overall rendering options, the `pbrtWorldBegin()` call marks the start of the description of the shapes, materials, and lights in the scene. It sets the current rendering state to `STATE_WORLD_BLOCK`, resets the CTMs to identity matrices, and enables all of the CTMs.

curTransform 1054
Inverse() 1021
namedCoordinateSystems 1055
ParamSet 1047
pbrtCamera() 1058
pbrtCoordSysTransform() 1056
pbrtWorldBegin() 1059
renderOptions 1056
RenderOptions::
CameraName 1058
RenderOptions::
CameraParams 1058
RenderOptions::
CameraToWorld 1058
STATE_WORLD_BLOCK 1052
TransformSet 1054

⟨*API Function Definitions*⟩ +≡
```
void pbrtWorldBegin() {
    VERIFY_OPTIONS("WorldBegin");
    currentApiState = STATE_WORLD_BLOCK;
    for (int i = 0; i < MAX_TRANSFORMS; ++i)
        curTransform[i] = Transform();
    activeTransformBits = ALL_TRANSFORMS_BITS;
    namedCoordinateSystems["world"] = curTransform;
}
```

B.3.1 HIERARCHICAL GRAPHICS STATE

As the scene's lights, geometry, and participating media are specified, a variety of attributes can be set as well. In addition to the CTMs, these include information about textures and the current material. When a geometric primitive or light source is then added to the scene, the current attributes are used when creating the corresponding object. These data are all known as the *graphics state.*

It is useful for a rendering API to provide some functionality for managing the graphics state. `pbrt` has API calls that allow the current graphics state to be managed with an *attribute stack*; the user can push the current set of attributes, make changes to their values, and then later pop back to the previously pushed attribute values. For example, a scene description file might contain the following:

```
Material "matte"
AttributeBegin
  Material "plastic"
  Translate 5 0 0
  Shape "sphere" "float radius" [1]
AttributeEnd
Shape "sphere" "float radius" [1]
```

The first sphere is affected by the translation and is bound to the plastic material, while the second sphere is matte and isn't translated. Changes to attributes made inside a `pbrtAttributeBegin()`/`pbrtAttributeEnd()` block are forgotten at the end of the block. Being able to save and restore attributes in this manner is a classic idiom for scene description in computer graphics.

The graphics state is stored in the `GraphicsState` structure. As was done previously with `RenderOptions`, we'll be adding members to it throughout this section.

⟨*API Local Classes*⟩ +≡
```
struct GraphicsState {
    ⟨Graphics State Methods 1067⟩
    ⟨Graphics State 1064⟩
};
```

activeTransformBits 1054
ALL_TRANSFORMS_BITS 1054
currentApiState 1052
curTransform 1054
GraphicsState 1059
MAX_TRANSFORMS 1054
namedCoordinateSystems 1055
pbrtAttributeBegin() 1060
pbrtAttributeEnd() 1060
RenderOptions 1056
STATE_WORLD_BLOCK 1052
Transform 76

⟨*API Local Classes*⟩ +≡
```
GraphicsState::GraphicsState() {
    ⟨GraphicsState Constructor Implementation 1064⟩
}
```

When `pbrtInit()` is called, the current graphics state is initialized to hold default values.

⟨*API Initialization*⟩ +≡ 1051
```
graphicsState = GraphicsState();
```

A `vector` of `GraphicsStates` is used as a stack to perform hierarchical state management. When `pbrtAttributeBegin()` is called, the current `GraphicsState` is copied and pushed onto this stack. `pbrtAttributeEnd()` then simply pops the state from this stack.

⟨*API Function Definitions*⟩ +≡
```
void pbrtAttributeBegin() {
    VERIFY_WORLD("AttributeBegin");
    pushedGraphicsStates.push_back(graphicsState);
    pushedTransforms.push_back(curTransform);
    pushedActiveTransformBits.push_back(activeTransformBits);
}
```

⟨*API Static Data*⟩ +≡
```
static GraphicsState graphicsState;
static vector<GraphicsState> pushedGraphicsStates;
static vector<TransformSet> pushedTransforms;
static vector<uint32_t> pushedActiveTransformBits;
```

`pbrtAttributeEnd()` also verifies that we do not have stack underflow by checking to see if there are any entries on the stack.

⟨*API Function Definitions*⟩ +≡
```
void pbrtAttributeEnd() {
    VERIFY_WORLD("AttributeEnd");
    if (!pushedGraphicsStates.size()) {
        Error("Unmatched pbrtAttributeEnd() encountered. "
              "Ignoring it.");
        return;
    }
    graphicsState = pushedGraphicsStates.back();
    pushedGraphicsStates.pop_back();
    curTransform = pushedTransforms.back();
    pushedTransforms.pop_back();
    activeTransformBits = pushedActiveTransformBits.back();
    pushedActiveTransformBits.pop_back();
}
```

We also provide the `pbrtTransformBegin()` and `pbrtTransformEnd()` calls. These functions are similar to `pbrtAttributeBegin()` and `pbrtAttributeEnd()`, except that they only push and pop the CTMs. We frequently want to apply a transformation to a texture, but since the list of named textures is stored in the graphics state, we cannot use

activeTransformBits 1054
curTransform 1054
Error() 1005
GraphicsState 1059
graphicsState 1060
pbrtAttributeBegin() 1060
pbrtAttributeEnd() 1060
pbrtInit() 1051
pbrtTransformBegin() 1061
pbrtTransformEnd() 1061
pushedActiveTransformBits 1060
pushedGraphicsStates 1060
pushedTransforms 1060
TransformSet 1054

pbrtAttributeBegin() to save the transformation matrix. Since the implementations of pbrtTransformBegin() and pbrtTransformEnd() are very similar to pbrtAttributeBegin() and pbrtAttributeEnd(), respectively, they are not shown here.

⟨*API Function Declarations*⟩ +≡
```
void pbrtTransformBegin();
void pbrtTransformEnd();
```

B.3.2 TEXTURE AND MATERIAL PARAMETERS

Recall that all of the materials in pbrt use textures to describe all of their parameters. For example, the diffuse color of the matte material class is always obtained from a texture, even if the material is intended to have a constant reflectivity (in which case a ConstantTexture is used).

Before a material can be created, it is necessary to create these textures to pass to the material creation procedures. Textures can be either explicitly created and later referred to by name or implicitly created on the fly to encapsulate a constant parameter. These two methods of texture creation are hidden by the TextureParams class.

⟨*TextureParams Declarations*⟩ ≡
```
class TextureParams {
public:
    ⟨TextureParams Public Methods 1061⟩
private:
    ⟨TextureParams Private Data 1061⟩
};
```

The TextureParams class contains associative arrays of previously defined named float and Spectrum textures, as well as two ParamSets that will be searched for named textures.

⟨*TextureParams Private Data*⟩ ≡ 1061
```
map<string, Reference<Texture<float> > > &floatTextures;
map<string, Reference<Texture<Spectrum> > > &spectrumTextures;
const ParamSet &geomParams, &materialParams;
```

⟨*TextureParams Public Methods*⟩ ≡ 1061
```
TextureParams(const ParamSet &geomp, const ParamSet &matp,
              map<string, Reference<Texture<float> > > &ft,
              map<string, Reference<Texture<Spectrum> > > &st)
    : floatTextures(ft), spectrumTextures(st),
      geomParams(geomp), materialParams(matp) {
}
```

Here we will show the code for finding a texture of Spectrum type; the code for finding a float texture is analogous. The TextureParams::GetSpectrumTexture() method takes a parameter name (e.g., "Kd") , as well as a default Spectrum value. If no texture has been explicitly specified for the parameter, a constant texture will be created that returns the default spectrum value.

ConstantTexture 520
ParamSet 1047
pbrtAttributeBegin() 1060
pbrtAttributeEnd() 1060
pbrtTransformBegin() 1061
pbrtTransformEnd() 1061
Reference 1011
Spectrum 263
Texture 519
TextureParams 1061
TextureParams::
GetSpectrumTexture() 1062

Finding the texture is performed in several stages; the order of these stages is crucial. First, the parameter list from the `Shape` for which a `Material` is being created is searched for a named reference to an explicitly defined texture. If no such texture is found, then the material parameters are searched. Finally, if no explicit texture has been found, the two parameter lists are searched in turn for supplied constant values. If no such constants are found, the default is used.

The order of these steps is crucial, because `pbrt` allows a shape to override individual elements of the material that is bound to it. For example, the user should be able to create a scene description that contains the lines

```
Material "matte" "color Kd" [ 1 0 0 ]
Shape "sphere" "color Kd" [ 0 1 0 ]
```

These two commands should create a green matte sphere. Because the shape's parameter list is searched first, the `Kd` parameter from the `Shape` command will be used when the `MatteMaterial` constructor is called.

⟨*TextureParams Method Definitions*⟩ ≡

```
Reference<Texture<Spectrum> >
TextureParams::GetSpectrumTexture(const string &n,
                                  const Spectrum &def) const {
    string name = geomParams.FindTexture(n);
    if (name == "") name = materialParams.FindTexture(n);
    if (name != "") {
        if (spectrumTextures.find(name) != spectrumTextures.end())
            return spectrumTextures[name];
        else
            Error("Couldn't find spectrum texture named \"%s\" "
                  "for parameter \"%s\"", name.c_str(), n.c_str());
    }
    Spectrum val = geomParams.FindOneSpectrum(n,
                                materialParams.FindOneSpectrum(n, def));
    return new ConstantTexture<Spectrum>(val);
}
```

Because an instance of the `TextureParams` class is passed to material creation routines that might need to access nontexture parameter values, we also provide ways to access the other parameter list types. These methods return parameters from the geometric parameter list, if found. Otherwise, the material parameter list is searched, and finally the default value is returned.

The `TextureParams::FindFloat()` method is shown here. The other access methods are similar and omitted.

⟨*TextureParams Public Methods*⟩ +≡ 1061

```
float FindFloat(const string &n, float d) const {
    return geomParams.FindOneFloat(n, materialParams.FindOneFloat(n, d));
}
```

ConstantTexture 520
Error() 1005
Material 483
MatteMaterial 484
ParamSet::FindOneFloat() 1050
ParamSet::FindOneSpectrum() 1050
ParamSet::FindTexture() 1050
Reference 1011
Shape 108
Spectrum 263
Texture 519
TextureParams 1061
TextureParams::FindFloat() 1062
TextureParams::geomParams 1061
TextureParams::materialParams 1061
TextureParams::spectrumTextures 1061

B.3.3 SURFACE AND MATERIAL DESCRIPTION

The pbrtTexture() method creates a named texture that can be referred to later. In addition to the texture name, its *type* is specified. Currently, pbrt supports only "float" and "color" as texture types. The supplied parameter list is used to create a TextureParams object, which will be passed to the desired texture's creation routine.

⟨*API Function Definitions*⟩ +≡
```
void pbrtTexture(const string &name, const string &type,
                 const string &texname, const ParamSet &params) {
    VERIFY_WORLD("Texture");
    TextureParams tp(params, params, graphicsState.floatTextures,
                     graphicsState.spectrumTextures);
    if (type == "float")  {
        ⟨Create float texture and store in floatTextures 1063⟩
    }
    else if (type == "color")  {
        ⟨Create color texture and store in spectrumTextures⟩
    }
    else
        Error("Texture type \"%s\" unknown.", type.c_str());
}
```

Creating the texture is simple. This function first checks to see if a texture of the same name and type already exists and issues a warning if so. Then, the MakeFloatTexture() routine calls the creation function for the appropriate Texture implementation, and the returned texture class is added to the GraphicsState::floatTextures associative array. The code for creating a color texture is similar and not shown.

⟨*Create* float *texture and store in* floatTextures⟩ ≡ 1063
```
if (graphicsState.floatTextures.find(name) !=
    graphicsState.floatTextures.end())
    Warning("Texture \"%s\" being redefined", name.c_str());
WARN_IF_ANIMATED_TRANSFORM("Texture");
Reference<Texture<float> > ft = MakeFloatTexture(texname,
                                                 curTransform[0], tp);
if (ft) graphicsState.floatTextures[name] = ft;
```

curTransform 1054
Error() 1005
graphicsState 1060
GraphicsState::floatTextures 1064
GraphicsState::spectrumTextures 1064
ParamSet 1047
pbrtTexture() 1063
Reference 1011
Texture 519
TextureParams 1061
TransformSet::IsAnimated() 1064
Warning() 1005
WARN_IF_ANIMATED_TRANSFORM() 1063

Not all of the types in pbrt that take transformations support animated transformations. Textures are one example; it's not worth the additional code complexity to support them, especially since the utility an animated texture transform brings isn't obvious. The WARN_IF_ANIMATED_TRANSFORM() macro warns if the CTMs are different, indicating animated transformations, for classes that don't actualy use more than a single transformation.

⟨*API Macros*⟩ +≡
```
#define WARN_IF_ANIMATED_TRANSFORM(func) \
do { if (curTransform.IsAnimated()) \
         Warning("Animated transformations set; ignoring for \"%s\"" \
                 "and using the start transform only", func); \
} while (false)
```

⟨*TransformSet Public Methods*⟩ +≡ 1054
```
bool IsAnimated() const {
    for (int i = 0; i < MAX_TRANSFORMS-1; ++i)
        if (t[i] != t[i+1]) return true;
    return false;
}
```

⟨*Graphics State*⟩ ≡ 1059
```
map<string, Reference<Texture<float> > > floatTextures;
map<string, Reference<Texture<Spectrum> > > spectrumTextures;
```

The current material is specified by a call to `pbrtMaterial()`. Its `ParamSet` is stored until a `Material` object needs to be created later when a shape is specified.

⟨*API Function Declarations*⟩ +≡
```
void pbrtMaterial(const string &name, const ParamSet &params);
```

⟨*Graphics State*⟩ +≡ 1059
```
ParamSet materialParams;
string material;
```

The default material for all objects in the scene is matte:

⟨*GraphicsState Constructor Implementation*⟩ ≡ 1060
```
material = "matte";
```

`pbrt` also supports the notion of creating a `Material` with a given set of parameters and then associating an arbitrary name with the combination of material and parameter settings. `pbrtMakeNamedMaterial()` creates such an association, and `pbrtNamedMaterial()` sets the current material and material parameters based on a previously defined named material.

⟨*API Function Declarations*⟩ +≡
```
void pbrtMakeNamedMaterial(const string &name, const ParamSet &params);
void pbrtNamedMaterial(const string &name);
```

⟨*Graphics State*⟩ +≡ 1059
```
map<string, Reference<Material> > namedMaterials;
string currentNamedMaterial;
```

B.3.4 LIGHT SOURCES

`pbrt`'s API provides two ways for the user to specify light sources for the scene. The first, `pbrtLightSource()`, defines a light source that doesn't have geometry associated with it (e.g., a point light or a directional light). The second, `pbrtAreaLightSource()`, specifies an area light source. All shape specifications that appear between an area light source call up to the end of the current attribute block are treated as emissive.

GraphicsState::material 1064
Material 483
MAX_TRANSFORMS 1054
ParamSet 1047
pbrtAreaLightSource() 1065
pbrtLightSource() 1065
pbrtMakeNamedMaterial() 1064
pbrtMaterial() 1064
pbrtNamedMaterial() 1064
Reference 1011
Spectrum 263
Texture 519
TransformSet::t 1054

⟨*API Function Definitions*⟩ +≡
```
void pbrtLightSource(const string &name, const ParamSet &params) {
    VERIFY_WORLD("LightSource");
    WARN_IF_ANIMATED_TRANSFORM("LightSource");
    Light *lt = MakeLight(name, curTransform[0], params);
    if (lt == NULL)
        Error("pbrtLightSource: light type \"%s\" unknown.", name.c_str());
    else
        renderOptions->lights.push_back(lt);
}
```

⟨*RenderOptions Public Data*⟩ +≡ 1056
```
vector<Light *> lights;
```

When an area light is specified via `pbrtAreaLightSource()`, it can't be created immediately since the shapes to follow are needed to define the light source's geometry. Therefore, this function just saves the name of the area light source type and the parameters given to it.

⟨*Graphics State*⟩ +≡ 1059
```
ParamSet areaLightParams;
string areaLight;
```

⟨*API Function Definitions*⟩ +≡
```
void pbrtAreaLightSource(const string &name,
                         const ParamSet &params) {
    VERIFY_WORLD("AreaLightSource");
    graphicsState.areaLight = name;
    graphicsState.areaLightParams = params;
}
```

B.3.5 SHAPES AND VOLUME REGIONS

The `pbrtShape()` function creates a new `Shape` object and adds it to the scene. This function is relatively complicated, in that it has to create area light sources for the shape if an area light is being defined, it has to handle animated and static shapes differently, and it also has to deal with creating object instances when needed.

⟨*API Function Definitions*⟩ +≡
```
void pbrtShape(const string &name, const ParamSet &params) {
    VERIFY_WORLD("Shape");
    Reference<Primitive> prim;
    AreaLight *area = NULL;
    if (!curTransform.IsAnimated()) {
        ⟨Create primitive for static shape 1066⟩
    } else {
        ⟨Create primitive for animated shape 1067⟩
    }
    ⟨Add primitive to scene or current instance 1068⟩
}
```

AreaLight 623
curTransform 1054
Error() 1005
graphicsState 1060
Light 606
MakeLight() 1073
ParamSet 1047
pbrtAreaLightSource() 1065
pbrtShape() 1065
Primitive 185
Reference 1011
renderOptions 1056
RenderOptions::lights 1065
Shape 108
TransformSet::
IsAnimated() 1064

Shapes that are animated are represented with TransformedPrimitives, which include extra functionality to use AnimatedTransforms, while shapes that aren't animated use GeometricPrimitives. Therefore, there are two code paths here for those two cases.

The static shape case is mostly a matter of creating the appropriate Shape and Material to make a GeometricPrimitive and then allocating a GeometricPrimitive. MakeShape() handles the details of creating the shape corresponding to the given shape name, passing the ParamSet along to its creation routine. The code below uses a TransformCache (defined shortly), which allocates and stores a single Transform pointer for each unique transformation that is passed to its Lookup() method. In this way, if many shapes in the scene have the same transformation matrix, a single Transform pointer can be shared among all of them.

⟨*Create primitive for static shape*⟩ ≡ 1065
```
Transform *obj2world, *world2obj;
transformCache.Lookup(curTransform[0], &obj2world, &world2obj);
Reference<Shape> shape = MakeShape(name, obj2world, world2obj,
    graphicsState.reverseOrientation, params);
if (!shape) return;
Reference<Material> mtl = graphicsState.CreateMaterial(params);
params.ReportUnused();
⟨Possibly create area light for shape 1067⟩
prim = new GeometricPrimitive(shape, mtl, area);
```

TransformCache is a small wrapper around an associative array from transformations to pairs of Transform pointers; the first pointer is equal to the transform and the second is its inverse. The Lookup() method just looks for the given transformation in the cache, allocates space for it and stores it and its inverse if not found, and returns the appropriate pointers.

⟨*TransformCache Private Data*⟩ ≡
```
map<Transform, std::pair<Transform *, Transform *> > cache;
MemoryArena arena;
```

⟨*API Static Data*⟩ +≡
```
static TransformCache transformCache;
```

The MakeShape() function takes the name of the shape to be created, the CTMs, and the ParamSet for the new shape. It calls an appropriate shape creation function based on the shape name provided. Here only the sphere creation code is shown; code for the rest of the shapes is equivalent.

⟨*Object Creation Function Definitions*⟩ ≡
```
Reference<Shape> MakeShape(const string &name,
        const Transform *object2world, const Transform *world2object,
        bool reverseOrientation, const ParamSet &paramSet) {
    Shape *s = NULL;
```

AnimatedTransform 96
CreateSphereShape() 1073
curTransform 1054
GeometricPrimitive 188
graphicsState 1060
GraphicsState::CreateMaterial() 1067
MakeShape() 1066
Material 483
MemoryArena 1015
ParamSet 1047
ParamSet::ReportUnused() 1051
Reference 1011
Shape 108
Transform 76
TransformCache 1066
transformCache 1066
TransformCache::Lookup() 1066
TransformedPrimitive 190

```
    if (name == "sphere")
        s = CreateSphereShape(object2world, world2object,
                              reverseOrientation, paramSet);
    ⟨Create remaining Shape types⟩
    paramSet.ReportUnused();
    return s;
}
```

The `Material` for the shape is created by the `MakeMaterial()` call; its implementation is analogous to that of `MakeShape()`. If the specified material cannot be found (usually due to a typo in the material name), a matte material is created and a warning is issued.

⟨*Graphics State Methods*⟩ ≡ 1059
```
Reference<Material> CreateMaterial(const ParamSet &params);
```

If an area light has been set in the current graphics state by `pbrtAreaLightSource()`, the new shape is an emitter and an `AreaLight` needs to be made for it.

⟨*Possibly create area light for shape*⟩ ≡ 1066
```
if (graphicsState.areaLight != "") {
    area = MakeAreaLight(graphicsState.areaLight, curTransform[0],
                         graphicsState.areaLightParams, shape);
}
```

If the transformation matrices are animated, the task is a little more complicated. A base `Shape` and `GeometricPrimitive` are first created. Recall from Section 4.1.2 that the `TransformedPrimitive` holds only a single `Primitive` and requires that it supports ray intersection queries; if the shape is not immediately intersectable, it is refined into intersectable primitives and a BVH is created, yielding a single primitive for the `TransformedPrimitive` constructor.

AreaLight 623
curTransform 1054
GeometricPrimitive 188
graphicsState 1060
GraphicsState::
areaLight 1065
GraphicsState::
areaLightParams 1065
MakeShape() 1066
Material 483
ParamSet 1047
pbrtAreaLightSource() 1065
Primitive 185
Primitive::
CanIntersect() 186
Reference 1011
Shape 108
TransformedPrimitive 190

⟨*Create primitive for animated shape*⟩ ≡ 1065
```
⟨Create initial Shape for animated shape 1068⟩
⟨Get animatedWorldToObject transform for shape 1068⟩
Reference<Primitive> baseprim = new GeometricPrimitive(shape, mtl, NULL);
if (!baseprim->CanIntersect()) {
    ⟨Refine animated shape and create BVH if more than one shape created 1068⟩
}
prim = new TransformedPrimitive(baseprim, animatedWorldToObject);
```

Because the `Shape` class doesn't handle animated transformations, here the initial `Shape` for animated primitives is created with an identity transform passed to the shape creation routine. All of the details related to the shape's transformation are managed with the `TransformedPrimitive` that ends up holding the shape. Animated transformations for light sources aren't currently supported in `pbrt`; thus, if an animated transform has been specified with an area light source, a warning is issued here.

⟨*Create initial* Shape *for animated shape*⟩ ≡ 1067
```
if (graphicsState.areaLight != "")
    Warning("Ignoring currently set area light when creating "
            "animated shape");
Transform *identity;
transformCache.Lookup(Transform(), &identity, NULL);
Reference<Shape> shape = MakeShape(name, identity, identity,
    graphicsState.reverseOrientation, params);
if (!shape) return;
Reference<Material> mtl = graphicsState.CreateMaterial(params);
params.ReportUnused();
```

Again the TransformCache is used, here to get transformations for the start and end time, which are then passed to the AnimatedTransform constructor.

⟨*Get* animatedWorldToObject *transform for shape*⟩ ≡ 1067
```
Transform *world2obj[2];
transformCache.Lookup(curTransform[0], NULL, &world2obj[0]);
transformCache.Lookup(curTransform[1], NULL, &world2obj[1]);
AnimatedTransform
     animatedWorldToObject(world2obj[0], renderOptions->transformStartTime,
                           world2obj[1], renderOptions->transformEndTime);
```

⟨*Refine animated shape and create BVH if more than one shape created*⟩ ≡ 1067
```
vector<Reference<Primitive> > refinedPrimitives;
baseprim->FullyRefine(refinedPrimitives);
if (refinedPrimitives.size() == 0) return;
if (refinedPrimitives.size() > 1)
    baseprim = new BVHAccel(refinedPrimitives);
else
    baseprim = refinedPrimitives[0];
```

If the user is in the middle of defining an object instance, pbrtObjectBegin() (defined in the following section) will have set the currentInstance member of renderOptions to point to a vector that is collecting the shapes that define the instance. In that case, the new shape is added to that array. Otherwise, the shape is stored in the RenderOptions::primitives array. This array will eventually be passed to the Scene constructor. If this primitive is an area light, it is also added to the RenderOptions::lights array, just as the pbrtLightSource() call also does.

⟨*RenderOptions Public Data*⟩ +≡ 1056
```
vector<Reference<Primitive> > primitives;
```

⟨*Add primitive to scene or current instance*⟩ ≡ 1065
```
if (renderOptions->currentInstance) {
    if (area)
        Warning("Area lights not supported with object instancing");
    renderOptions->currentInstance->push_back(prim);
}
```

AnimatedTransform 96
BVHAccel 209
curTransform 1054
graphicsState 1060
GraphicsState::areaLight 1065
GraphicsState::CreateMaterial() 1067
MakeShape() 1066
Material 483
ParamSet::ReportUnused() 1051
pbrtLightSource() 1065
pbrtObjectBegin() 1069
Primitive 185
Primitive::FullyRefine() 186
Reference 1011
renderOptions 1056
RenderOptions::currentInstance 1069
RenderOptions::lights 1065
RenderOptions::primitives 1068
RenderOptions::transformEndTime 1057
RenderOptions::transformStartTime 1057
Scene 22
Shape 108
Transform 76
TransformCache 1066
transformCache 1066
TransformCache::Lookup() 1066
Warning() 1005

```
else {
    renderOptions->primitives.push_back(prim);
    if (area != NULL) {
        renderOptions->lights.push_back(area);
    }
}
```

B.3.6 OBJECT INSTANCING

All shapes that are specified between a `pbrtObjectBegin()` and `pbrtObjectEnd()` pair are used to create a named object instance (see the discussion of object instancing and the `TransformedPrimitive` class in Section 4.1.2). `pbrtObjectBegin()` sets `RenderOptions::currentInstance` so that subsequent `pbrtShape()` calls can add the shape to this instance's vector of primitive references. This function also pushes the graphics state, so that any changes made to the CTMs or other state while defining the instance don't last beyond the instance definition.

⟨*API Function Definitions*⟩ +≡

```
void pbrtObjectBegin(const string &name) {
    VERIFY_WORLD("ObjectBegin");
    pbrtAttributeBegin();
    if (renderOptions->currentInstance)
        Error("ObjectBegin called inside of instance definition");
    renderOptions->instances[name] = vector<Reference<Primitive> >();
    renderOptions->currentInstance = &renderOptions->instances[name];
}
```

⟨*RenderOptions Public Data*⟩ +≡ 1056

```
map<string, vector<Reference<Primitive> > > instances;
vector<Reference<Primitive> > *currentInstance;
```

⟨*RenderOptions Constructor Implementation*⟩ +≡ 1056

```
currentInstance = NULL;
```

⟨*API Function Definitions*⟩ +≡

```
void pbrtObjectEnd() {
    VERIFY_WORLD("ObjectEnd");
    if (!renderOptions->currentInstance)
        Error("ObjectEnd called outside of instance definition");
    renderOptions->currentInstance = NULL;
    pbrtAttributeEnd();
}
```

When an instance is used in the scene, the instance's vector of `Primitives` needs to be found in the `RenderOptions::instances` map, a `TransformedPrimitive` created, and the instance added to the scene. Note that the `TransformedPrimitive` constructor takes the current transformation matrix from the time when `pbrtObjectInstance()` is called. The instance's complete world transformation is the composition of the CTM when it is instantiated with the CTM when it was originally created.

Error() 1005
pbrtAttributeBegin() 1060
pbrtAttributeEnd() 1060
pbrtObjectBegin() 1069
pbrtObjectEnd() 1069
pbrtObjectInstance() 1070
pbrtShape() 1065
Primitive 185
Reference 1011
renderOptions 1056
RenderOptions::
currentInstance 1069
RenderOptions::
instances 1069
TransformedPrimitive 190

pbrtObjectInstance() first does some error checking to make sure that the instance is not being used inside the definition of another instance and also that the named instance has been defined. The error checking is simple and not shown here.

⟨*API Function Definitions*⟩ +≡
```
void pbrtObjectInstance(const string &name) {
    VERIFY_WORLD("ObjectInstance");
    ⟨Object instance error checking⟩
    vector<Reference<Primitive> > &in = renderOptions->instances[name];
    if (in.size() == 0) return;
    if (in.size() > 1 || !in[0]->CanIntersect()) {
        ⟨Refine instance Primitives and create aggregate 1070⟩
    }
    Transform *world2instance[2];
    transformCache.Lookup(curTransform[0], NULL, &world2instance[0]);
    transformCache.Lookup(curTransform[1], NULL, &world2instance[1]);
    AnimatedTransform animatedWorldToInstance(world2instance[0],
        renderOptions->transformStartTime,
        world2instance[1], renderOptions->transformEndTime);
    Reference<Primitive> prim =
        new TransformedPrimitive(in[0], animatedWorldToInstance);
    renderOptions->primitives.push_back(prim);
}
```

If there is more than one shape in an instance, or if the instance's shape must be refined before it can be intersected, then an aggregate needs to be built for it. This must be done here rather than in the TransformedPrimitive constructor so that the resulting aggregate will be reused if this instance is used multiple times in the scene.

⟨*Refine instance* Primitives *and create aggregate*⟩ ≡ 1070
```
Reference<Primitive> accel =
     MakeAccelerator(renderOptions->AcceleratorName,
                     in, renderOptions->AcceleratorParams);
if (!accel) accel = MakeAccelerator("bvh", in, ParamSet());
if (!accel) Severe("Unable to create \"bvh\" accelerator");
in.erase(in.begin(), in.end());
in.push_back(accel);
```

B.3.7 WORLD END AND RENDERING

When pbrtWorldEnd() is called, the scene has been fully specified and rendering can begin. This routine makes sure that there aren't excess graphics state structures pushed on the state stack (issuing a warning if so), creates the Scene and Renderer objects, and then calls the Renderer::Render() method.

AnimatedTransform 96
curTransform 1054
ParamSet 1047
pbrtObjectInstance() 1070
Primitive 185
Primitive::
CanIntersect() 186
Reference 1011
Renderer 24
Renderer::Render() 24
renderOptions 1056
RenderOptions::
instances 1069
RenderOptions::
primitives 1068
Severe() 1005
Transform 76
transformCache 1066
TransformedPrimitive 190

⟨*API Function Definitions*⟩ +≡
```
void pbrtWorldEnd() {
    VERIFY_WORLD("WorldEnd");
    ⟨Ensure there are no pushed graphics states 1071⟩
    ⟨Create scene and render 1071⟩
    ⟨Clean up after rendering 1071⟩
}
```

If there are graphics states and/or transformations remaining on the respective stacks, a warning is issued for each one:

⟨*Ensure there are no pushed graphics states*⟩ ≡ 1071
```
while (pushedGraphicsStates.size()) {
    Warning("Missing end to pbrtAttributeBegin()");
    pushedGraphicsStates.pop_back();
    pushedTransforms.pop_back();
}
while (pushedTransforms.size()) {
    Warning("Missing end to pbrtTransformBegin()");
    pushedTransforms.pop_back();
}
```

Now the `RenderOptions::MakeRenderer()` and `RenderOptions::MakeScene()` methods can create the corresponding objects based on the settings provided by the user, and rendering can occur.

⟨*Create scene and render*⟩ ≡ 1071
```
Renderer *renderer = renderOptions->MakeRenderer();
Scene *scene = renderOptions->MakeScene();
if (scene && renderer) renderer->Render(scene);
TasksCleanup();
delete renderer;
delete scene;
```

Once rendering is complete, the API transitions back to the "options block" rendering state, prints out any statistics gathered by probes during rendering, and clears the CTMs and named coordinate systems so that the next frame, if any, starts with a clean slate.

⟨*Clean up after rendering*⟩ ≡ 1071
```
currentApiState = STATE_OPTIONS_BLOCK;
ProbesPrint(stdout);
for (int i = 0; i < MAX_TRANSFORMS; ++i)
    curTransform[i] = Transform();
activeTransformBits = ALL_TRANSFORMS_BITS;
namedCoordinateSystems.erase(namedCoordinateSystems.begin(),
                             namedCoordinateSystems.end());
```

activeTransformBits 1054
ALL_TRANSFORMS_BITS 1054
currentApiState 1052
curTransform 1054
MAX_TRANSFORMS 1054
namedCoordinateSystems 1055
pushedGraphicsStates 1060
pushedTransforms 1060
Renderer 24
Renderer::Render() 24
renderOptions 1056
RenderOptions::MakeRenderer() 1072
RenderOptions::MakeScene() 1072
Scene 22
STATE_OPTIONS_BLOCK 1052
TasksCleanup() 1041
Transform 76
Warning() 1005

Creating the Scene object is mostly a matter of creating the Aggregate for all of the primitives and calling the Scene constructor. The MakeAccelerator() function isn't included here; it's similar in structure to MakeShape() as far as using the string passed to it to determine which accelerator construction function to call.

⟨*API Function Definitions*⟩ +≡
```
Scene *RenderOptions::MakeScene() {
    ⟨Initialize volumeRegion from volume region(s) 1072⟩
    Primitive *accelerator = MakeAccelerator(AcceleratorName,
        primitives, AcceleratorParams);
    if (!accelerator)
        accelerator = MakeAccelerator("bvh", primitives, ParamSet());
    if (!accelerator)
        Severe("Unable to create \"bvh\" accelerator.");
    Scene *scene = new Scene(accelerator, lights, volumeRegion);
    ⟨Erase primitives, lights, and volume regions from RenderOptions 1072⟩
    return scene;
}
```

If more than one VolumeRegion was specified, an AggregateVolume is created to store them.

⟨*Initialize* volumeRegion *from volume region(s)*⟩ ≡ 1072
```
VolumeRegion *volumeRegion;
if (volumeRegions.size() == 0)
    volumeRegion = NULL;
else if (volumeRegions.size() == 1)
    volumeRegion = volumeRegions[0];
else
    volumeRegion = new AggregateVolume(volumeRegions);
```

After the scene has been created, RenderOptions clears the vectors of primitives, lights, and volume regions. This ensures that if a subsequent scene is defined then the scene description from this frame isn't inadvertently included.

⟨*Erase primitives, lights, and volume regions from* RenderOptions⟩ ≡ 1072
```
primitives.erase(primitives.begin(), primitives.end());
lights.erase(lights.begin(), lights.end());
volumeRegions.erase(volumeRegions.begin(), volumeRegions.end());
```

Renderer creation in the MakeRenderer() method is again similar to how shapes and other named objects are created; the string name is used to dispatch to an object-specific creation function.

⟨*RenderOptions Public Methods*⟩ ≡ 1056
```
Renderer *MakeRenderer() const;
```

Aggregate 192
AggregateVolume 597
MakeShape() 1066
ParamSet 1047
Primitive 185
Renderer 24
RenderOptions 1056
RenderOptions::lights 1065
RenderOptions::primitives 1068
Scene 22
VolumeRegion 587

B.4 ADDING NEW OBJECT IMPLEMENTATIONS

We will briefly review the overall process that pbrt uses to create instances of implementations of the various abstract interface classes like Shapes, Cameras, SurfaceIntegrators, etc., at run time. We will focus on the details for the Shape class, since the other types are handled similarly.

When the API needs to create a shape, it has the string name of the shape and the ParamSet that represents the corresponding information in the input file. These need to be used together to create a specific instance of the named shape. MakeShape() has a series of if tests to determine which shape creation function to call; the one for Spheres, CreateSphereShape(), is shown here:

⟨*Sphere Method Definitions*⟩ +≡

```
Sphere *CreateSphereShape(const Transform *o2w, const Transform *w2o,
        bool reverseOrientation, const ParamSet &params) {
    float radius = params.FindOneFloat("radius", 1.f);
    float zmin = params.FindOneFloat("zmin", -radius);
    float zmax = params.FindOneFloat("zmax", radius);
    float phimax = params.FindOneFloat("phimax", 360.f);
    return new Sphere(o2w, w2o, reverseOrientation, radius,
                      zmin, zmax, phimax);
}
```

The appropriate named parameter values are extracted from the parameter list, sensible defaults are used for ones not present, and the appropriate values are passed to the Sphere constructor. As another alternative, we could have written the Sphere constructor to just take a ParamSet as a parameter and extracted the parameters there. We followed this approach instead in order to make it easier to create spheres for other uses without having to create a full ParamSet to specify the parameters to it.

Thus, adding a new implementation to pbrt requires adding the new source files to the build process so they are compiled and linked to the executable, modifying the appropriate creation function (MakeShape(), MakeLight(), etc.) in core/api.cpp to look for the new type's name and call its creation function, in addition to implementing the creation function (like CreateSphereShape()) that extracts parameters and calls the object's constructor, returning a new instance of the object.

Camera 302
CreateSphereShape() 1073
MakeShape() 1066
ParamSet 1047
ParamSet::
FindOneFloat() 1050
Shape 108
Sphere 115
SurfaceIntegrator 740
Transform 76

FURTHER READING

Foley et al. (1990) have discussed hierarchical modeling and retained mode versus immediate mode rendering APIs in depth; their chapter on these issues (Chapter 7) is a good starting point for more information about these topics. The *OpenGL Programming Guide* (Woo et al. 1999) is a good reference for learning the OpenGL interface. The RenderMan® API is described in a number of books (Upstill 1989; Apodaca and Gritz 2000) and in its specification document, which is available online (Pixar Animation Studios 2000). A different approach to rendering APIs is taken by the *mental ray* rendering system, which

exposes much more internal information about the system's implementation to users, allowing it to be extensible in more ways than is possible in other systems. See Driemeyer and Herken (2002) for further information about it.

EXERCISES

B.1 `pbrt`'s scene file parser is written using the standard `lex` and `yacc` tools. While these are an easy way to develop such a parser, carefully implemented handwritten parsers can be substantially more efficient. Replace `pbrt`'s parser with a handwritten parser that parses the same file format. Measure the change in performance with your parser. Profile `pbrt` when rendering scenes with large scene description files. What fraction of execution time is spent in parsing?

B.2 `pbrt`'s scene description format makes it easy for other programs to export `pbrt` scenes and for users to edit the scene files to make small adjustments to them. However, for complex scenes (such as the ecosystem scene in Figure 4.1), the large text files that are necessary to describe them can take a long time to parse.

Investigate extensions to `pbrt` to support compact binary file formats for scene files. One possible approach is to support binary encoding for large meshes: since most of the complexity in detailed scenes comes from large polygon and subdivision surface meshes, providing specialized encodings for just those shapes may be almost as effective as a binary encoding for the entire scene description format and has the advantage of not requiring that the scene file parser be rewritten. You could extend `TriangleMesh` and/or `LoopSubdiv` to take an optional string parameter that gives the filename for a binary file that holds some or all of the mesh vertex positions and normals and the array of integers that describes which triangles use which vertices.

A more complex alternative is to develop a complete binary encoding for the text scene description file format. Each directive in the file (`WorldBegin`, `Shape`, etc.) could be encoded with a unique byte-sized value, with string values following them (such as `"sphere"`) encoded as NUL-terminated strings, and then their parameter lists encoded with a similar mix of strings and binary-encoded integer and floating-point values.

B.3 Another approach for reducing renderer startup time is to support a binary representation of internal data structures that can be written to disk. For example, for complex scenes, creating the ray acceleration aggregates may take more time than the initial parsing of the scene file. An alternative is to modify the system to have the capability of dumping out a representation of the acceleration structure and all of the primitives inside it after it is first created. The resulting file could then be subsequently read back into memory much more quickly than rebuilding the data structure from scratch. However, because C++ doesn't have native support for saving arbitrary objects to disk and then reading them back during a subsequent execution of the program (a capability known as *serialization* or *pickling* in other languages), adding this feature effectively requires extending many of the objects in `pbrt` to support this capability on their own.

LoopSubdiv 151
TriangleMesh 135

One additional advantage of this approach is that substantial amounts of computation can be invested in creating high-quality acceleration structures, with the knowledge that this cost doesn't need to be paid each time the scene is loaded into memory. This approach is more complex than the one in Exercise B.2, but should give better performance.

❷ B.4 The material assigned to object instances in `pbrt` is the current material when the instance was defined inside the `pbrtObjectBegin()`/`pbrtObjectEnd()` block. This can be inconvenient if the user wants to use the same geometry multiple times in a scene, giving it a different material. Fix the API and the implementation of the `TransformedPrimitive` class so that the material assigned to instances is the current material when the instance is instantiated, or, if the user hasn't set a material, the current material when the instance was created.

❸ B.5 Generalize `pbrt`'s API for specifying animation; the current implementation only allows the user to provide two transformation matrices, only at the start and end of a fixed time range. For specifying more complex motion, a more flexible approach may be useful. One improvement is to allow the user to specify an arbitrary number of *keyframe* transformations, each associated with an arbitrary time.

More generally, the system could be extended to support transformations that are explicit functions of time. For example, a rotation could be described with an expression of the form `Rotate (time * 2 + 1) 0 0 1` to describe a time-varying rotation about the z axis. Extend `pbrt` to support a more general matrix animation scheme and render images showing the visual improvement. What is the performance impact of your changes for scenes with animated objects that don't need the generality of your improvements?

❸ B.6 Extend `pbrt`'s API to have some retained mode semantics so that animated sequences of images can be rendered without needing to respecify the entire scene for each frame. Make sure that it is possible to remove some objects from the scene, add others, modify objects' materials and transformations from frame to frame, and so on.

❷ B.7 In the current implementation, a unique `TransformedPrimitive` is created for each `Shape` with an animated transformation. If many shapes have exactly the same animated transformation, this turns out to be a poor choice. Consider the difference between a million-triangle mesh with an animated transformation versus a million independent triangles, all of which happen to have the same animated transformation.

In the first case, all of the triangles in the mesh are stored in a single instance of a `TransformedPrimitive` with an animated transformation. If a ray intersects the conservative bounding box that encompasses all of the object's motion over the frame time, then it is transformed to the mesh's object space according to the interpolated transformation at the ray's time. At this point, the intersection computation is no different than the intersection test with a static primitive; the only overhead due to the animation is from the larger bounding box and rays

`pbrtObjectBegin()` 1069
`pbrtObjectEnd()` 1069
`Shape` 108
`TransformedPrimitive` 190

that hit the bounding box but not the animated primitive and the extra computation for matrix interpolation and transforming each ray once, according to its time.

In the second case, each triangle is stored in its own `TransformedPrimitive`, all of which happen to have the same `AnimatedTransform`. Each instance of `TransformedPrimitive` will have a large bounding box to encompass each triangle's motion, giving the acceleration structure a difficult set of inputs to deal with: many primitives with substantially overlapping bounding boxes. The impact on ray–primitive intersection efficiency will be high: the ray will be redundantly transformed many times by what happens to be the same recomputed interpolated transformation, and many intersection tests will be performed due to the large bounding boxes. Overall performance will be much worse than the first case.

To address this case, modify the code that implements the `pbrt` API calls so that if many independent primitives are provided with the same animated transformation, they're all collected into a single acceleration structure with a single animated transformation. What is the performance improvement for the worst-case outlined above? Is there an impact for more typical scenes with animated primitives?

AnimatedTransform 96
TransformedPrimitive 190

INDEX OF FRAGMENTS

Bold numbers indicate the first page of a fragment definition, ***bold italic*** numbers indicate an extension of the definition, and roman numbers indicate a use of the fragment.

⟨*Accumulate contribution of direction ω to transfer coefficients*⟩, **973**, 973
⟨*Accumulate irradiance value from nearby photons*⟩, 832, **833**
⟨*Accumulate M_o from leaf node*⟩, **911**, 911
⟨*Accumulate outgoing radiance due to reflected irradiance*⟩, 816, **817**
⟨*AdaptiveSampler Declarations*⟩, **386**
⟨*AdaptiveSampler Method Definitions*⟩, **387**, ***387***, ***388***
⟨*AdaptiveSampler Private Data*⟩, **386**, 386
⟨*Add bias compensation ray contribution to radiance sum*⟩, 783, **784**
⟨*Add computed irradiance value to cache*⟩, 791, **796**
⟨*Add contribution from* `VirtualLight vl`⟩, 780, **782**
⟨*Add contribution of each light source*⟩, 43, **45**
⟨*Add contribution of escaped ray, if any*⟩, 847, **849**, 849
⟨*Add contribution of* `light` *due to scattering at* `p`⟩, **884**, 884
⟨*Add contributions of constant segments before/after samples*⟩, 268, **269**
⟨*Add emitted light from vertex if appropriate*⟩, 847, **848**, 850
⟨*Add* `InfiniteAreaLight` *texel's contribution to SH coefficients*⟩, **945**, 945
⟨*Add* `IrradiancePoint ip` *to interior octree node*⟩, 899, **900**, 900
⟨*Add* `IrradiancePoint` *to leaf octree node*⟩, 899, **900**
⟨*Add light contribution from BSDF sampling*⟩, **750**, 750
⟨*Add light sample contribution to MC estimate of SH coefficients*⟩, **943**, 943
⟨*Add light's contribution to reflected radiance*⟩, 749, **750**
⟨*Add photon to unordered array of photons*⟩, 822, **823**
⟨*Add primitive to already-allocated voxel*⟩, 200, **201**
⟨*Add primitive to overlapping voxels*⟩, 199, **200**
⟨*Add primitive to scene or current instance*⟩, 1065, **1068**
⟨*Add primitives to grid voxels*⟩, 197, **199**
⟨*Add splat value at pixel*⟩, 412, **413**
⟨*Adjust normal based on orientation and handedness*⟩, 102, **103**
⟨*Advance to first relevant wavelength segment*⟩, 268, **269**
⟨*Advance to next best-candidate sample table position*⟩, 383, **384**
⟨*Advance to next child node, possibly enqueue other child*⟩, 242, **244**
⟨*Advance to next pixel for sampling for* `AdaptiveSampler`⟩, 387, **388**
⟨*Advance to next pixel for stratified sampling*⟩, 351, **357**
⟨*Advance to next voxel*⟩, 205, **207**
⟨*Advance to sample at* `t0` *and update* `T`⟩, 880, **881**
⟨*Aggregate Declarations*⟩, **192**
⟨*AggregateTest Method Definitions*⟩, **247**
⟨*AggregateTest Private Data*⟩, **247**
⟨*AggregateVolume Private Data*⟩, **597**, 597
⟨*Allocate and initialize* `sample`⟩, **27**, 27
⟨*Allocate and request samples for sampling all lights*⟩, **743**, 743, 774, 788, 803
⟨*Allocate and request samples for sampling one light*⟩, 743, **744**
⟨*Allocate* `BSDF`*, possibly doing bump mapping with* `bumpMap`⟩, **485**, 485, 487, 489
⟨*Allocate* `BucketInfo` *for SAH partition buckets*⟩, **219**, 219
⟨*Allocate film image storage*⟩, 405, **407**
⟨*Allocate* `IrradianceSample`*, get write lock, add to octree*⟩, 796, **797**
⟨*Allocate kd-tree node and continue recursively*⟩, 1030, **1031**
⟨*Allocate local variables for* `isect` *and* `T` *if needed*⟩, 34, **35**
⟨*Allocate* `LoopSubdiv` *vertices and faces*⟩, **151**, 151
⟨*Allocate new voxel and store primitive in it*⟩, 200, **201**

⟨*Allocate next level of children in mesh tree*⟩, 161, **162**
⟨*Allocate octree node if needed and continue recursive traversal*⟩, 1024, **1025**
⟨*Allocate SH coefficient vector pointers for sample points*⟩, 959, **960**
⟨*Allocate space for samples and intersections*⟩, 30, **31**
⟨*Allocate storage for sample pointers*⟩, **345**, 345
⟨*Allocate storage for sample values*⟩, 345, **346**
⟨*Allocate working memory for kd-tree construction*⟩, 232, **236**, ***240***
⟨*Allow integrators to do preprocessing for the scene*⟩, **27**, 27
⟨*AnimatedTransform Method Definitions*⟩, **97**, ***99***, ***100***
⟨*AnimatedTransform Private Data*⟩, **96**
⟨*AnimatedTransform Public Methods*⟩, **96**, ***100***
⟨*Anisotropic Public Methods*⟩, **458**, 458, ***460***
⟨*API Cleanup*⟩, 1052, **1053**, ***1056***
⟨*API Function Declarations*⟩, **1055**, ***1058***, ***1061***, ***1064***
⟨*API Function Definitions*⟩, **1051**, ***1052***, ***1054***, ***1055***, ***1056***, ***1057***, ***1058***, ***1059***, ***1060***, ***1063***, ***1065***, ***1069***, ***1070***, ***1071***, ***1072***
⟨*API Global Variables*⟩, **1051**
⟨*API Initialization*⟩, 1051, **1053**, ***1056***, ***1060***
⟨*API Local Classes*⟩, **1054**, ***1054***, ***1056***, ***1059***, ***1060***
⟨*API Macros*⟩, **1053**, ***1055***, ***1063***
⟨*API Static Data*⟩, **1052**, ***1054***, ***1055***, ***1056***, ***1060***, ***1066***
⟨*Apply boundary rule for even vertex*⟩, 162, **165**
⟨*Apply box filter to checkerboard region*⟩, 549, **551**
⟨*Apply edge rules to compute new vertex position*⟩, **167**, 167
⟨*Apply large step to ith camera* `PathSample`⟩, 840, **841**
⟨*Apply one-ring rule for even vertex*⟩, 162, **163**
⟨*Apply SH definitions to compute final (l, m) values*⟩, 936, **941**
⟨*Apply* `sWeights` *to zoom in s direction*⟩, 532, **534**
⟨*AreaLight Interface*⟩, 623, **625**
⟨*Bail out if we found a hit closer than the current node*⟩, **242**, 242
⟨*BBox Method Definitions*⟩, **71**, ***74***, ***194***
⟨*BBox Public Data*⟩, 70, **71**
⟨*BBox Public Methods*⟩, **70**, 70, ***71***, ***72***, ***73***
⟨*Begin the KdTree building process*⟩, **1029**, 1029
⟨*BestCandidate Sampling Constants*⟩, **380**
⟨*BestCandidateSampler Declarations*⟩, **381**
⟨*BestCandidateSampler Method Definitions*⟩, **383**, ***383***
⟨*BestCandidateSampler Private Data*⟩, **381**, 381, ***383***, ***384***
⟨*BestCandidateSampler Public Methods*⟩, **381**, 381
⟨*BilerpTexture Declarations*⟩, **522**
⟨*BilerpTexture Private Data*⟩, **522**, 522
⟨*BilerpTexture Public Methods*⟩, **522**, 522, ***523***
⟨*Blinn Public Methods*⟩, 456, **457**
⟨*BlockedArray Private Data*⟩, 1017, **1018**
⟨*BlockedArray Public Methods*⟩, 1017, **1018**, ***1019***, ***1020***
⟨*BoundEdge Public Methods*⟩, **235**, 235, ***237***
⟨*Box Filter Declarations*⟩, **396**
⟨*Box Filter Method Definitions*⟩, **396**
⟨*BRDFToBTDF Public Methods*⟩, **431**, ***431***, 431
⟨*BSDF Declarations*⟩, **428**, ***478***, ***482***, ***705***, ***706***
⟨*BSDF Inline Functions*⟩, **426**, ***426***, ***427***, ***695***
⟨*BSDF Inline Method Definitions*⟩, **479**
⟨*BSDF Method Definitions*⟩, **478**, ***481***, ***706***
⟨*BSDF Private Data*⟩, 478, **479**, ***479***, ***488***
⟨*BSDF Private Methods*⟩, 478, **483**
⟨*BSDF Public Data*⟩, 478, **479**
⟨*BSDF Public Methods*⟩, 478, **479**, ***480***, ***482***, ***708***
⟨*BSDFSample Public Methods*⟩, **705**, ***705***, 705
⟨*BSSRDF Declarations*⟩, **598**
⟨*BSSRDF Private Data*⟩, 598, **599**
⟨*BSSRDF Public Methods*⟩, **598**, 598, ***599***
⟨*Build BVH from* `primitives`⟩, 209, **210**
⟨*Build kd-tree for accelerator*⟩, 229, **232**
⟨*Build kd-trees for indirect and caustic photons*⟩, 804, **813**
⟨*BVHAccel Local Declarations*⟩, **211**, ***211***, ***216***, ***217***, ***222***, ***226***
⟨*BVHAccel Method Definitions*⟩, **209**, ***213***, ***224***
⟨*BVHAccel Private Data*⟩, **210**, ***223***
⟨*BVHBuildNode Public Methods*⟩, 211, **212**, ***212***
⟨*BxDF Declarations*⟩, **428**, ***431***, ***436***, ***437***, ***439***, ***440***, ***444***, ***447***, ***454***, ***456***, ***458***, ***460***, ***464***, ***467***
⟨*BxDF Interface*⟩, 428, **429**, ***429***, ***430***
⟨*BxDF Local Definitions*⟩, **466**
⟨*BxDF Method Definitions*⟩, **431**, ***432***, ***437***, ***441***, ***444***, ***447***, ***451***, ***454***, ***455***, ***461***, ***462***, ***465***, ***466***, ***469***, ***694***, ***695***, ***696***, ***699***, ***700***, ***701***, ***703***, ***704***
⟨*BxDF Public Data*⟩, 428, **429**
⟨*BxDF Utility Functions*⟩, **435**, ***435***
⟨*Camera Declarations*⟩, **302**, ***305***
⟨*Camera Interface*⟩, **302**, 302, ***303***
⟨*Camera Method Definitions*⟩, **303**, ***305***
⟨*Camera Public Data*⟩, **302**, 302
⟨*Check for intersection in current voxel and advance to next*⟩, **205**, 205
⟨*Check for intersections inside leaf node*⟩, 242, **244**
⟨*Check for ray intersection against x and y slabs*⟩, **226**, 226
⟨*Check one primitive inside leaf node*⟩, **244**, 244
⟨*Check ray against BVH node*⟩, **225**, 225
⟨*Check ray against overall grid bounds*⟩, 202, **203**
⟨*Check sample against crop window, goto* `again` *if outside*⟩, 383, **385**
⟨*Checkerboard2DTexture Private Data*⟩, **547**, 547, ***548***
⟨*Checkerboard2DTexture Public Methods*⟩, **547**, 547, ***548***
⟨*Checkerboard3DTexture Private Data*⟩, 552, **553**
⟨*Checkerboard3DTexture Public Methods*⟩, **552**, 552, ***553***
⟨*CheckerboardTexture Declarations*⟩, **547**, ***552***
⟨*Choose level of detail for EWA lookup and perform EWA filtering*⟩, 540, **541**
⟨*Choose light and call* `EstimateDirect()` *for Metropolis vertex*⟩, **848**, 848
⟨*Choose light and sample ray to start light path*⟩, **846**, 846
⟨*Choose light source to trace virtual light path from*⟩, **777**, 777
⟨*Choose light to shoot photon from*⟩, **807**, 807
⟨*Choose point on disk oriented toward infinite light direction*⟩, 714, **715**
⟨*Choose random rays,* `rayAccel` *and* `rayAll` *for testing*⟩, **247**, 247
⟨*Choose ray direction for testing accelerator*⟩, 247, **248**
⟨*Choose ray epsilon for testing accelerator*⟩, 247, **248**

⟨*Choose ray origin for testing accelerator*⟩, 247, **248**
⟨*Choose split axis position for interior node*⟩, 234, **236**
⟨*Choose split direction and partition data*⟩, 1030, **1031**
⟨*Choose which axis to split along*⟩, **236**, 236
⟨*Choose which* BxDF *to sample*⟩, 706, **707**
⟨*Clamp ellipse eccentricity if too large*⟩, **540**, 540
⟨*Classify primitives with respect to split*⟩, 234, **239**
⟨*Clean up after rendering*⟩, **1071**, 1071
⟨*Clean up after rendering and store final image*⟩, 27, **29**
⟨*ClosePhoton Public Methods*⟩, **822**, ***822***, 822
⟨*CoefficientSpectrum Protected Data*⟩, **264**, 264
⟨*CoefficientSpectrum Public Methods*⟩, **264**, ***264***, 264, ***265***, ***266***
⟨*Compare contrast of sample differences to threshold*⟩, 388, **389**
⟨*Compute* (u, v) *offsets at auxiliary points*⟩, 507, **508**
⟨*Compute acceptance probability for proposed sample*⟩, 857, **860**
⟨*Compute* α' *for RGB component, compute scattering properties*⟩, **913**, 913
⟨*Compute auxiliary intersection points with plane*⟩, **507**, 507
⟨*Compute average value of given SPD over* i *th sample's range*⟩, **267**, 267
⟨*Compute barycentric coordinates for point*⟩, 146, **147**
⟨*Compute bound of primitive centroids, choose split dimension* dim⟩, 213, **214**
⟨*Compute bounding box of region used to generate random rays*⟩, **247**, 247
⟨*Compute bounds and choose grid resolution*⟩, 197, **198**
⟨*Compute bounds for kd-tree construction*⟩, **232**, 232
⟨*Compute bounds of all primitives in BVH node*⟩, **213**, 213
⟨*Compute bounds of data from* start *to* end⟩, **1031**, 1031
⟨*Compute bump-mapped differential geometry*⟩, 495, **496**
⟨*Compute caustic lighting for photon map integrator*⟩, **818**
⟨*Compute cell indices for dots*⟩, **559**, 559
⟨*Compute closed-form box-filtered* Checkerboard2DTexture *value*⟩, 548, **549**
⟨*Compute coefficients for fundamental forms*⟩, **122**, 122, 128
⟨*Compute conditional sampling distribution for* $\tilde{v}$⟩, **672**, 672
⟨*Compute contribution between camera and light vertices*⟩, **851**, 851
⟨*Compute contribution for random sample for MLT bootstrapping*⟩, 855, **856**
⟨*Compute contribution of proposed sample*⟩, 857, **860**
⟨*Compute coordinate system for sphere sampling*⟩, 720, **721**
⟨*Compute cosine of cone surrounding projection directions*⟩, 616, **617**
⟨*Compute cosine term of Oren-Nayar model*⟩, **451**, 451
⟨*Compute cost for split at* i *th edge*⟩, 237, **238**
⟨*Compute cost of all splits for* axis *to find best*⟩, 236, **237**
⟨*Compute costs for splitting after each bucket*⟩, 219, **221**
⟨*Compute current voxel for axis*⟩, **204**, 204
⟨*Compute cylinder* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$⟩, **128**, 128
⟨*Compute cylinder* $\partial\mathrm{p}/\partial u$ *and* $\partial\mathrm{p}/\partial v$⟩, **128**, 128
⟨*Compute cylinder hit point and* ϕ⟩, **127**, 127, 128
⟨*Compute deltas for triangle partial derivatives*⟩, **143**, 143
⟨*Compute differential changes in origin for ortho camera rays*⟩, 306, **307**
⟨*Compute differential changes in origin for perspective camera rays*⟩, 310, **312**
⟨*Compute differential reflected directions*⟩, 512, **513**
⟨*Compute direct illumination for Metropolis path vertex*⟩, 847, **848**, 850
⟨*Compute direct lighting for* DirectLightingIntegrator *integrator*⟩, **744**
⟨*Compute direct lighting for irradiance cache*⟩, **789**
⟨*Compute* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$ *from fundamental form coefficients*⟩, **122**, 122, 128
⟨*Compute double spherical integral for BSDF matrix*⟩, 978, **979**
⟨*Compute ellipse coefficients to bound EWA filter region*⟩, 541, **542**
⟨*Compute ellipse minor and major axes*⟩, **540**, 540
⟨*Compute emission-only source term at* p⟩, 880, **882**
⟨*Compute emitted and reflected light at ray intersection point*⟩, 42, **43**
⟨*Compute emitted light if ray hit an area light source*⟩, **43**, 43
⟨*Compute environment camera ray direction*⟩, **319**, 319
⟨*Compute estimate error term and possibly use sample*⟩, 793, **794**
⟨*Compute exitant radiance from photons for glossy surface*⟩, 831, **832**
⟨*Compute exitant radiance* Lindir *using radiance photons*⟩, **824**, 824, 827
⟨*Compute falloff inside spotlight cone*⟩, **614**, 614
⟨*Compute film image extent*⟩, 405, **406**
⟨*Compute final average value for probe and cleanup*⟩, 963, **965**
⟨*Compute final coefficients* c_o *using BSDF matrix*⟩, 981, **982**
⟨*Compute first barycentric coordinate*⟩, 141, **142**
⟨*Compute Fresnel reflectance for dielectric*⟩, 437, **438**
⟨*Compute general sphere weight*⟩, **722**, 722
⟨*Compute glossy BSDF matrix for PRT*⟩, **980**, 980
⟨*Compute gradient weights*⟩, 555, **556**
⟨*Compute image resampling weights for* i *th texel*⟩, 532, **533**
⟨*Compute incident direction by reflecting about* ω_{h}⟩, 696, **697**, 699
⟨*Compute incident radiance from radiance probe coefficients*⟩, 966, **967**
⟨*Compute incident radiance sample from* light*, update SH* coeffs⟩, **943**, 943
⟨*Compute* index *into measured BRDF tables*⟩, **469**, 469
⟨*Compute indices of refraction for dielectric*⟩, **438**, 438
⟨*Compute indices* whThetaIndex*,* wdThetaIndex*,* wdPhiIndex⟩, 469, **470**
⟨*Compute indirect illumination with virtual lights*⟩, **780**
⟨*Compute indirect lighting for irradiance cache*⟩, **789**
⟨*Compute indirect lighting for photon map integrator*⟩, **818**
⟨*Compute* InfiniteAreaLight *ray PDF*⟩, 729, **730**
⟨*Compute initial parametric range of ray inside kd-tree extent*⟩, **240**, 240
⟨*Compute integral of product of incident radiance and transfer function*⟩, 972, **973**

⟨*Compute integral of step function at x_i*⟩, **648**, 648
⟨*Compute integrator samples for best-candidate sample*⟩, 383, **385**
⟨*Compute interpolated matrix as product of interpolated components*⟩, 99, **100**
⟨*Compute intersection distance along ray*⟩, 118, **119**
⟨*Compute intersections using accelerator and exhaustive testing*⟩, 247, **248**
⟨*Compute irradiance at current point*⟩, 791, **795**
⟨*Compute irradiance sample's contribution extent and bounding box*⟩, 796, **797**
⟨*Compute irradiance values at sample points*⟩, **897**, 897
⟨*Compute* `i` *th candidate point* `p` *in cell's bounding box*⟩, 963, **964**
⟨*Compute K_l^m coefficients*⟩, 936, **939**
⟨*Compute Legendre polynomial values for* $\cos\theta$⟩, **936**, 936
⟨*Compute light power CDF for photon shooting*⟩, 804, **805**
⟨*Compute* `L[current]` *for initial sample*⟩, **858**, 858
⟨*Compute $m = 0$ Legendre values using recurrence*⟩, 936, **938**
⟨*Compute $m = 1, \ldots, l - 2$ values using Legendre recurrence*⟩, 936, **939**
⟨*Compute $m = l$ edge using Legendre recurrence*⟩, 936, **938**
⟨*Compute $m = l - 1$ edge using Legendre recurrence*⟩, 936, **939**
⟨*Compute marginal sampling distribution $p[\tilde{v}]$*⟩, 672, **673**
⟨*Compute MIPMap level for trilinear filtering*⟩, 536, **537**
⟨*Compute MIS weight for BSDF-sampled gather ray*⟩, 824, **825**
⟨*Compute MIS weight for photon-sampled gather ray*⟩, **827**, 827
⟨*Compute Monte Carlo estimate of* `i` *th sample for transfer matrix*⟩, 976, **977**
⟨*Compute M_o at node if error is low enough*⟩, **911**, 911
⟨*Compute new odd edge vertices*⟩, 162, **166**
⟨*Compute new transformation matrix* `M` *without translation*⟩, **97**, 97
⟨*Compute next digit of radical inverse*⟩, **362**, 362
⟨*Compute next matrix* `Rnext` *in series*⟩, **98**, 98
⟨*Compute noise cell coordinates and offsets*⟩, **555**, 555
⟨*Compute norm of difference between* `R` *and* `Rnext`⟩, **98**, 98
⟨*Compute number of octaves for antialiased FBm*⟩, 561, **562**, 564
⟨*Compute number of* `SamplerRendererTask`*s to create for rendering*⟩, 28, **29**
⟨*Compute number of specular vertices for each path length*⟩, 849, **850**
⟨*Compute odd vertex on* `k` *th edge*⟩, 166, **167**
⟨*Compute offset along CDF segment*⟩, **649**, 649
⟨*Compute offset positions and evaluate displacement texture*⟩, **495**, 495
⟨*Compute offset rays for* `PerspectiveCamera` *ray differentials*⟩, **313**
⟨*Compute overall PDF with all matching* `BxDF`*s*⟩, 706, **707**
⟨*Compute parametric distance along ray to split plane*⟩, **242**, 242
⟨*Compute PDF for photon-sampling of direction* `wi`⟩, 825, **827**, 827
⟨*Compute PDF for sampled infinite light direction*⟩, 727, **728**
⟨*Compute PDF for sampled offset*⟩, **649**, 649
⟨*Compute PDF for ω_i from anisotropic distribution*⟩, 699, 700, **701**
⟨*Compute PDF for ω_i from Blinn distribution*⟩, 696, **699**, 699
⟨*Compute perfect specular reflection direction*⟩, **441**, 441
⟨*Compute permutation tables for each base*⟩, 367, **368**
⟨*Compute pixel spacing in world space at intersection point*⟩, 789, **790**
⟨*Compute plane intersection for disk*⟩, **131**, 131
⟨*Compute point on plane of focus*⟩, 315, **318**
⟨*Compute probe coordinates and offsets for lookup point*⟩, **966**, 966
⟨*Compute projective camera screen transformations*⟩, 305, **306**
⟨*Compute projective camera transformations*⟩, **305**, 305
⟨*Compute proposed mutation to current sample*⟩, 857, **859**
⟨*Compute quadratic cylinder coefficients*⟩, **127**, 127
⟨*Compute quadratic sphere coefficients*⟩, 117, **118**, 123
⟨*Compute quadratic* `t` *values*⟩, 118, **119**
⟨*Compute radiance along path using path tracing*⟩, 845, **846**, 846
⟨*Compute radiance along paths using bidirectional path tracing*⟩, **846**, 846
⟨*Compute radiance for radiance photon* `i`⟩, **816**, 816
⟨*Compute random points on surfaces of scene*⟩, 959, **962**
⟨*Compute randomly permuted table of pixel indices for large steps*⟩, 858, **859**
⟨*Compute range of radiance photons to process in task*⟩, **816**, 816
⟨*Compute raster sample from table*⟩, 383, **385**
⟨*Compute ray differential* `rd` *for specular reflection*⟩, 46, **512**
⟨*Compute* `rayEpsilon` *for quadric intersection*⟩, 117, **123**, 127, 131
⟨*Compute reflectance* `SampledSpectrum` *with* `rgb[0]` *as minimum*⟩, **278**, 278
⟨*Compute reflected light at camera path vertex*⟩, 849, **850**
⟨*Compute reflected lighting using radiance probes*⟩, **966**, 966
⟨*Compute reflected radiance due to irradiance and BSDF*⟩, 790, **791**
⟨*Compute reflected radiance using diffuse PRT*⟩, **972**
⟨*Compute reflected radiance with glossy PRT at point*⟩, **981**
⟨*Compute reflection for radiance probes integrator*⟩, **966**
⟨*Compute region in which to compute incident radiance probes*⟩, **963**, 963
⟨*Compute representation of depth-first traversal of BVH tree*⟩, 210, **223**
⟨*Compute sample patterns for single scattering samples*⟩, **884**
⟨*Compute sample's raster extent*⟩, **408**, 408
⟨*Compute sampled half-angle vector ω_h for Blinn distribution*⟩, 696, **697**
⟨*Compute samples for emitted rays from lights*⟩, 776, **777**
⟨*Compute sampling distributions for rows and columns of image*⟩, 725, **727**
⟨*Compute sampling rate in each dimension*⟩, **959**, 959
⟨*Compute scalar-valued image* `img` *from environment map*⟩, 725, **727**
⟨*Compute scale* `S` *using rotation and original matrix*⟩, 97, **98**
⟨*Compute scene bounds and initialize probe integrators*⟩, **959**, 959
⟨*Compute second barycentric coordinate*⟩, 141, **142**
⟨*Compute SH coefficients of incident radiance at point* `p`⟩, 963, **964**

⟨*Compute SH radiance transfer matrix at point and SH coefficients*⟩, **981**, 981
⟨*Compute sine and tangent terms of Oren-Nayar model*⟩, **451**, 451
⟨*Compute single-scattering source term at* p⟩, **884**
⟨*Compute* sin $i\phi$ *and* cos $i\phi$ *using recurrence*⟩, 940, **941**
⟨*Compute* sin ϕ *and* cos ϕ *values*⟩, 936, **940**
⟨*Compute* `sint` *using Snell's law*⟩, **438**, 438
⟨*Compute sphere* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$⟩, 120, **122**
⟨*Compute sphere* $\partial\mathrm{p}/\partial u$ *and* $\partial\mathrm{p}/\partial v$⟩, 120, **121**
⟨*Compute sphere hit position and* ϕ⟩, 117, **119**, 120, 123
⟨*Compute squared radius and filter texel if inside ellipse*⟩, 543, **544**
⟨*Compute sum of octaves of noise for FBm*⟩, 561, **563**
⟨*Compute sum of octaves of noise for turbulence*⟩, 564, **565**
⟨*Compute* `t` *to intersection point*⟩, 141, **142**
⟨*Compute tangents of boundary face*⟩, 171, **173**
⟨*Compute tangents of interior face*⟩, 171, **172**
⟨*Compute texel* (s, t) *accounting for boundary conditions*⟩, **535**, 535
⟨*Compute texel* (s, t) *in* s*-zoomed image*⟩, **534**, 534
⟨*Compute texture coordinate differentials for sphere* (u, v) *mapping*⟩, 516, **517**
⟨*Compute texture differentials for 2D identity mapping*⟩, **515**, 515
⟨*Compute the ellipse's* (s, t) *bounding box in texture space*⟩, 541, **542**
⟨*Compute total area of shapes in* `ShapeSet` *and area CDF*⟩, **723**
⟨*Compute total number of sample values needed*⟩, **345**, 345
⟨*Compute transmitted ray direction*⟩, 444, **445**
⟨*Compute triangle partial derivatives*⟩, 141, **143**
⟨*Compute trilinear interpolation of weights*⟩, 555, **558**
⟨*Compute valence of boundary vertex*⟩, 158, **159**
⟨*Compute valence of interior vertex*⟩, **158**, 158
⟨*Compute value of BSDF for sampled direction*⟩, 706, **708**
⟨*Compute* $\mathbf{s}_1$⟩, **141**, 141
⟨*Compute vertex tangents on limit surface*⟩, 161, **171**
⟨*Compute virtual light's tentative contribution* `Llight`⟩, 780, **782**
⟨*Compute voxel coordinates and offsets for* `Pobj`⟩, 593, **594**
⟨*Compute voxel widths and allocate voxels*⟩, 197, **199**
⟨*Compute weight for bidirectional path,* `pathWt`⟩, 851, **852**
⟨*Compute* ω_h *and transform* ω_i *to halfangle coordinate system*⟩, **469**, 469
⟨*Compute world-to-object transformation for instance*⟩, 190, **191**
⟨*Compute* x *and* y *pixel sample range for sub-window*⟩, 341, **342**
⟨*Compute XYZ matching functions for* `SampledSpectrum`⟩, 271, **272**
⟨*ConstantTexture Declarations*⟩, **520**
⟨*ConstantTexture Public Methods*⟩, **520**, 520
⟨*Convert EWA coordinates to appropriate scale for level*⟩, **541**, 541
⟨*Convert image to RGB and compute final pixel values*⟩, **412**, 412
⟨*Convert infinite light sample point to direction*⟩, 727, **728**
⟨*Convert leaf node to interior node, redistribute points*⟩, **900**, 900
⟨*Convert light sample weight to solid angle measure*⟩, **717**, 717
⟨*Convert pixel XYZ color to RGB*⟩, **412**, 412
⟨*Convert reflectance spectrum to RGB*⟩, 277, **278**
⟨*Convert texels to type* `Tmemory` *and create* `MIPMap`⟩, 525, **526**
⟨*Convolve incident radiance to compute irradiance function*⟩, 966, **968**
⟨*Copy integrator samples into* `samples[i]`⟩, **376**, 376
⟨*Copy photon directions to local array*⟩, 819, **823**
⟨*Cosine-sample the hemisphere, flipping the direction if necessary*⟩, **694**, 694, 701
⟨*Create and initialize new odd vertex*⟩, **167**, 167
⟨*Create and launch* `SamplerRendererTask`*s for rendering image*⟩, 27, **28**
⟨*Create* `float` *texture and store in* `floatTextures`⟩, **1063**, 1063
⟨*Create initial* `Shape` *for animated shape*⟩, 1067, **1068**
⟨*Create leaf* `BVHBuildNode`⟩, **213**, 213, 215, 221
⟨*Create leaf if no good splits were found*⟩, 234, **239**
⟨*Create leaf node of kd-tree if we've reached the bottom*⟩, **1030**, 1030
⟨*Create* `mipmap` *for* `GonioPhotometricLight`⟩, 619, **620**
⟨*Create octree of clustered irradiance samples*⟩, 897, **898**
⟨*Create one-valued* `MIPMap`⟩, 525, **527**
⟨*Create primitive for animated shape*⟩, 1065, **1067**
⟨*Create primitive for static shape*⟩, 1065, **1066**
⟨*Create* `ProjectionLight` *MIP-map*⟩, **616**, 616
⟨*Create reader-writer mutex for grid*⟩, 197, **205**
⟨*Create scene and render*⟩, **1071**, 1071
⟨*Create scene bounding sphere to catch rays that leave the scene*⟩, 890, **891**, 962
⟨*Create virtual light and sample new ray for path*⟩, 777, **778**
⟨*Create virtual light at ray intersection point*⟩, 778, **779**
⟨*Creater interior flattened BVH node*⟩, **224**, 224
⟨*CreateRadianceProbes Method Definitions*⟩, **959**, ***963***
⟨*CreateRadianceProbes Private Data*⟩, **958**
⟨*CreateRadianceProbes Public Methods*⟩, **958**
⟨*Cylinder Declarations*⟩, **124**
⟨*Cylinder Method Definitions*⟩, **125**, ***126***, ***127***, ***129***, ***719***
⟨*Cylinder Private Data*⟩, 124, **126**
⟨*CylindricalMapping2D Private Methods*⟩, **517**, 517
⟨*Declare basic* `MLTTask` *variables and prepare for sampling*⟩, 857, **858**
⟨*Declare common path integration variables*⟩, **767**, 767
⟨*Declare common variables for* `SurfacePointTask::Run()`⟩, **892**, 892
⟨*Declare local variables for* `PhotonShootingTask`⟩, **806**, ***806***, 806, ***815***
⟨*Declare local variables used for rendering loop*⟩, 30, **31**
⟨*Declare sample parameters for light source sampling*⟩, 742, **743**, 788, 803

⟨*Declare shared variables for photon shooting*⟩, **804**, 804, ***815***
⟨*Declare shared variables for Poisson point generation*⟩, **890**, 890
⟨*Declare variables for storing and computing MLT samples*⟩, **858**, 858
⟨*Decorrelate sample dimensions*⟩, 352, **353**
⟨*DensityRegion Protected Data*⟩, 591, **592**
⟨*DensityRegion Public Methods*⟩, 591, **592**, ***592***
⟨*Deposit photon at surface*⟩, 808, **810**
⟨*Determine bases b_i and their sum*⟩, **367**, 367
⟨*Determine how many tiles to use in each dimension,* `nx` *and* `ny`⟩, 341, **342**
⟨*Determine which children the item overlaps*⟩, 1024, **1026**
⟨*Determine which octree child node* `p` *is inside*⟩, **1027**, 1027
⟨*DifferentialGeometry Declarations*⟩, **102**
⟨*DifferentialGeometry Method Definitions*⟩, **102**, ***505***
⟨*DifferentialGeometry Public Data*⟩, **102**, ***102***, 102, ***505***
⟨*DiffuseAreaLight Declarations*⟩, **625**
⟨*DiffuseAreaLight Method Definitions*⟩, **626**, ***627***, ***718***
⟨*DiffuseAreaLight Protected Data*⟩, 625, **626**
⟨*DiffuseAreaLight Public Methods*⟩, 625, **626**, ***627***
⟨*DiffusePRTIntegrator Method Definitions*⟩, **972**, ***972***
⟨*DiffusePRTIntegrator Private Data*⟩, **972**
⟨*DiffusionReflectance Data*⟩, **907**, 907
⟨*DiffusionReflectance Public Methods*⟩, **907**, 907, ***908***
⟨*DipoleSubsurfaceIntegrator Helper Declarations*⟩, **898**
⟨*DipoleSubsurfaceIntegrator Local Declarations*⟩, **898**, ***907***
⟨*DipoleSubsurfaceIntegrator Method Definitions*⟩, **897**, ***911***
⟨*DipoleSubsurfaceIntegrator Private Data*⟩, **888**, ***897***, ***898***
⟨*DipoleSubsurfaceIntegrator Public Methods*⟩, **887**
⟨*DirectLightingIntegrator Declarations*⟩, **742**, ***742***
⟨*DirectLightingIntegrator Method Definitions*⟩, **743**
⟨*DirectLightingIntegrator Private Data*⟩, **742**, ***742***, 742
⟨*Discard directions behind projection light*⟩, **617**, 617
⟨*Disk Declarations*⟩, **129**
⟨*Disk Method Definitions*⟩, **130**, ***131***, ***132***, ***719***
⟨*Disk Private Data*⟩, 129, **130**
⟨*DistantLight Declarations*⟩, **621**
⟨*DistantLight Method Definitions*⟩, **621**, ***622***, ***623***, ***714***
⟨*DistantLight Private Data*⟩, 621, **622**
⟨*Distribution1D Private Data*⟩, **648**, 648
⟨*Distribution1D Public Methods*⟩, **648**, 648, ***649***, ***650***
⟨*Distribution2D Private Data*⟩, **672**, 672, ***673***
⟨*Distribution2D Public Methods*⟩, 672, **673**, ***673***
⟨*Do atomic add with gcc x86 inline assembly*⟩, 1036, **1037**
⟨*Do bias compensation for bounding geometry term*⟩, 780, **783**
⟨*Do emission-only volume integration in* `vr`⟩, **880**, 880
⟨*Do large step mutation of* `cameraSample`⟩, **840**, 840
⟨*Do one-bounce final gather for photon map*⟩, 818, **819**
⟨*Do photon map lookup at intersection point*⟩, **831**, 831
⟨*Do trilinear interpolation to compute SH coefficients at point*⟩, **967**, 967
⟨*DotsTexture Declarations*⟩, **559**
⟨*DotsTexture Private Data*⟩, **559**, 559
⟨*DotsTexture Public Methods*⟩, **559**, ***559***, 559
⟨*dtrace.d**⟩, **1008**
⟨*Either create leaf or split primitives at selected SAH bucket*⟩, 219, **221**
⟨*EmissionIntegrator Declarations*⟩, **877**
⟨*EmissionIntegrator Method Definitions*⟩, **877**, ***879***, ***880***
⟨*EmissionIntegrator Private Data*⟩, **877**, ***877***, 877
⟨*EmissionIntegrator Public Methods*⟩, **877**, 877
⟨*Enqueue* `secondChild` *in todo list*⟩, **244**, 244
⟨*Ensure there are no pushed graphics states*⟩, **1071**, 1071
⟨*EnvironmentCamera Declarations*⟩, **318**
⟨*EnvironmentCamera Method Definitions*⟩, **319**
⟨*EnvironmentCamera Public Methods*⟩, 318, **319**
⟨*Erase primitives, lights, and volume regions from* `RenderOptions`⟩, **1072**, 1072
⟨*Estimate direct lighting from* `light` *samples*⟩, **745**, 745
⟨*Estimate indirect lighting with irradiance cache*⟩, **789**, 789
⟨*Estimate one term of ρ_{hd}*⟩, 703, **704**
⟨*Estimate one term of ρ_{hh}*⟩, **704**, 704
⟨*Estimate reflected radiance due to incident photons*⟩, **831**, 831
⟨*Estimate screen space change in* `p` *and* (u, v)⟩, 505, **507**
⟨*Evaluate BSDF at hit point*⟩, **43**, 43
⟨*Evaluate BSSRDF and possibly compute subsurface scattering*⟩, **908**
⟨*Evaluate filter value at* (x, y) *pixel*⟩, 409, **410**
⟨*Evaluate irradiance function and accumulate reflection*⟩, 966, **969**
⟨*Evaluate outgoing radiance function for ω_{o} and add to* `L`⟩, 981, **982**
⟨*Evaluate radiance along camera ray*⟩, 32, **33**
⟨*Evaluate single check if filter is entirely inside one of them*⟩, 549, **550**
⟨*Evaluate textures for* `MatteMaterial` *material and allocate BRDF*⟩, 485, **486**
⟨*Exit task if enough photons have been found*⟩, 806, **813**
⟨*ExponentialDensity Declarations*⟩, **595**
⟨*ExponentialDensity Private Data*⟩, 595, **596**
⟨*ExponentialDensity Public Methods*⟩, **595**, 595, ***596***
⟨*Extract rotation* `R` *from transformation matrix*⟩, 97, **98**
⟨*Extract translation* `T` *from transformation matrix*⟩, **97**, 97
⟨*FBmTexture Declarations*⟩, **565**
⟨*FBmTexture Private Data*⟩, 565, **566**
⟨*FBmTexture Public Methods*⟩, 565, **566**, ***566***
⟨*Figure out which η is incident and which is transmitted*⟩, 444, **445**
⟨*Fill in* `DifferentialGeometry` *from triangle hit*⟩, 141, **145**
⟨*Film Declarations*⟩, **403**
⟨*Film Interface*⟩, **403**, ***403***, 403, ***404***
⟨*Film Method Definitions*⟩, **404**
⟨*Film Public Data*⟩, **403**, 403
⟨*Filter Declarations*⟩, **393**
⟨*Filter four texels from finer level of pyramid*⟩, **536**, 536
⟨*Filter Interface*⟩, **393**, 393, ***394***
⟨*Filter Public Data*⟩, 393, **394**
⟨*Find (u, v) sample coordinates in infinite light texture*⟩, **727**, 727
⟨*Find bucket to split at that minimizes SAH metric*⟩, 219, **221**

⟨*Find camera ray for* sample[i]⟩, **32**, 32
⟨*Find closest intersection of ray with shapes in* ShapeSet⟩, **723**, 723
⟨*Find indirect photons around point for importance sampling*⟩, 819, **821**
⟨*Find light and BSDF sample values for direct lighting estimate*⟩, 745, **746**
⟨*Find next vertex of path*⟩, 767, **769**
⟨*Find parametric representation of cylinder hit*⟩, 127, **128**
⟨*Find parametric representation of disk hit*⟩, 131, **132**
⟨*Find parametric representation of sphere hit*⟩, 117, **120**
⟨*Find quadratic discriminant*⟩, **118**, 118
⟨*Find ray after shifting one pixel in the x direction*⟩, **303**, 303
⟨*Find ray intersection with scene geometry or bounding sphere*⟩, 893, **894**
⟨*Find* stepAxis *for stepping to next voxel*⟩, 207, **208**
⟨*Find surrounding CDF segments and* offset⟩, **649**, 649, 650
⟨*Find voxel extent of primitive*⟩, 199, **200**
⟨*Find* voxelsPerUnitDist *for grid*⟩, 198, **199**
⟨*Finish vertex initialization*⟩, 151, **158**
⟨*floatfile.h**⟩, **1006**
⟨*Follow path* i *from light to create virtual lights*⟩, 776, **777**
⟨*Follow photon path through scene and record intersections*⟩, 807, **808**
⟨*Follow photon paths for a block of samples*⟩, 806, **807**
⟨*Follow ray path and attempt to deposit candidate sample points*⟩, 892, **893**
⟨*Follow ray through BVH nodes to find primitive intersections*⟩, 224, **225**
⟨*Free* MemoryArena *memory from computing image sample values*⟩, 32, **33**
⟨*Fresnel Interface*⟩, **436**, 436
⟨*FresnelBlend Private Data*⟩, 460, **461**
⟨*FresnelBlend Public Methods*⟩, 460, **462**
⟨*FresnelConductor Public Methods*⟩, 436, **437**
⟨*FresnelDielectric Public Methods*⟩, **437**, 437
⟨*Function Definitions*⟩, **2**, ***3***
⟨*Gaussian Filter Declarations*⟩, **397**
⟨*Gaussian Filter Method Definitions*⟩, **398**
⟨*GaussianFilter Private Data*⟩, **397**, 397
⟨*GaussianFilter Public Methods*⟩, **397**, 397
⟨*GaussianFilter Utility Functions*⟩, 397, **398**
⟨*Generate camera path from camera path samples*⟩, **845**, 845
⟨*Generate camera rays and compute radiance along rays*⟩, **32**, 32
⟨*Generate initial stratified samples into* sampleBuf *memory*⟩, **352**, 352
⟨*Generate lens, time, and integrator samples for* HaltonSampler⟩, 364, **366**
⟨*Generate LHS samples along diagonal*⟩, **356**, 356
⟨*Generate low-discrepancy pixel samples*⟩, 375, **376**
⟨*Generate* photonRay *from light source and initialize* alpha⟩, 807, **808**
⟨*Generate random ray from intersection point*⟩, 893, **894**
⟨*Generate random sample and path radiance for MLT bootstrapping*⟩, 855, **856**
⟨*Generate raster and camera samples*⟩, **309**, 309, 312
⟨*Generate sample with Halton sequence and reject if outside image extent*⟩, 364, **365**
⟨*Generate stratified camera samples for* (xPos, yPos)⟩, 351, **352**
⟨*Generate stratified samples for integrators*⟩, 354, **356**
⟨*GeometricPrimitive Declarations*⟩, **187**
⟨*GeometricPrimitive Method Definitions*⟩, **188**, ***189***
⟨*GeometricPrimitive Private Data*⟩, 187, **188**
⟨*GeometricPrimitive Public Methods*⟩, 187, **188**
⟨*Geometry Declarations*⟩, **57**, ***63***, ***65***, ***66***, ***69***, ***70***
⟨*Geometry Inline Functions*⟩, **59**, ***60***, ***61***, ***62***, ***63***, ***65***, ***66***, ***291***, ***292***
⟨*Get* animatedWorldToObject *transform for shape*⟩, 1067, **1068**
⟨*Get irradiance* E *and average incident direction* wi *at point* p⟩, 790, **791**
⟨*Get new block of memory for* MemoryArena⟩, **1016**, 1016
⟨*Get next free node from* nodes *array*⟩, **233**, 233
⟨*Get node children pointers for ray*⟩, 242, **243**
⟨*Get one-ring vertices for boundary vertex*⟩, 164, **165**
⟨*Get one-ring vertices for interior vertex*⟩, **164**, 164
⟨*Get* outgoingBSDFSample *for sampling new path direction*⟩, 768, **769**
⟨*Get radiance probe coefficients around lookup point*⟩, **966**, 966
⟨*Get samples from* Sampler *and update image*⟩, 30, **32**
⟨*Get sub-*Sampler *for* SamplerRendererTask⟩, 30, **31**
⟨*Get* SurfacePoint*s for translucent objects in scene*⟩, **897**, 897
⟨*Get triangle vertices in* p1, p2, *and* p3⟩, 139, **140**, 140, 141, 145, 719
⟨*Global Constants*⟩, **1000**, ***1002***, ***1012***
⟨*Global Forward Declarations*⟩, **263**, ***1051***
⟨*Global Include Files*⟩, **999**, ***1000***, ***1005***
⟨*Global Inline Functions*⟩, **118**, ***1000***, ***1001***, ***1002***, ***1005***
⟨*Global Macros*⟩, **1009**
⟨*GlossyPRTIntegrator Method Definitions*⟩, **980**
⟨*GlossyPRTIntegrator Private Data*⟩, **980**, ***980***
⟨*GlossyPRTIntegrator Public Methods*⟩, **980**
⟨*GonioPhotometricLight Declarations*⟩, **618**
⟨*GonioPhotometricLight Method Definitions*⟩, **619**, ***621***
⟨*GonioPhotometricLight Private Data*⟩, 618, **620**
⟨*GonioPhotometricLight Public Methods*⟩, 618, **621**
⟨*Grab next node to process from todo list*⟩, 242, **245**
⟨*Graphics State*⟩, 1059, **1064**, ***1064***, ***1065***
⟨*Graphics State Methods*⟩, 1059, **1067**
⟨*GraphicsState Constructor Implementation*⟩, 1060, **1064**
⟨*GridAccel Declarations*⟩, **196**
⟨*GridAccel Method Definitions*⟩, **197**, ***202***, ***206***
⟨*GridAccel Private Data*⟩, 196, **198**, ***198***, ***199***, ***201***, ***205***
⟨*GridAccel Private Methods*⟩, 196, **200**, ***200***, ***201***
⟨*GridAccel Public Methods*⟩, 196, **208**
⟨*HaltonSampler Method Definitions*⟩, **364**
⟨*HaltonSampler Private Data*⟩, **364**
⟨*Hand kd-tree node to processing function*⟩, 1032, **1033**
⟨*Handle boundary conditions for matrix interpolation*⟩, **99**, 99
⟨*Handle cases with out-of-bounds range or single sample only*⟩, **268**, 268
⟨*Handle degenerate parametric mapping*⟩, **147**, 147
⟨*Handle first region of disk*⟩, **668**, 668

⟨*Handle new edge*⟩, **156**, 156
⟨*Handle photon/surface intersection*⟩, **808**, 808
⟨*Handle previously seen edge*⟩, **156**, 156
⟨*Handle ray that doesn't intersect any geometry*⟩, 34, **35**
⟨*Handle ray that leaves the scene during path generation*⟩, 843, **844**
⟨*Handle ray with negative direction for voxel stepping*⟩, 204, **205**
⟨*Handle ray with positive direction for voxel stepping*⟩, **204**, 204
⟨*Handle total internal reflection*⟩, 438, **439**
⟨*Handle total internal reflection for transmission*⟩, 445, **446**
⟨*Handle zero determinant for triangle partial derivative matrix*⟩, **143**, 143
⟨*HomogeneousVolumeDensity Declarations*⟩, **589**
⟨*HomogeneousVolumeDensity Private Data*⟩, 589, **590**
⟨*HomogeneousVolumeDensity Public Methods*⟩, 589, **590**, ***590***, ***591***
⟨*IGIIntegrator Local Structures*⟩, **779**
⟨*IGIIntegrator Method Definitions*⟩, **774**, ***776***
⟨*IGIIntegrator Private Data*⟩, **773**, ***774***, ***775***, ***779***
⟨*ImageFilm Declarations*⟩, **404**
⟨*ImageFilm Method Definitions*⟩, **405**, ***408***, ***411***, ***412***
⟨*ImageFilm Private Data*⟩, 404, **405**, ***406***, ***407***
⟨*ImageIO Declarations*⟩, **1004**, ***1004***
⟨*ImageTexture Declarations*⟩, **524**
⟨*ImageTexture Method Definitions*⟩, **524**, ***525***, ***527***
⟨*ImageTexture Private Data*⟩, **524**, 524, ***525***
⟨*ImageTexture Private Methods*⟩, 524, **526**, ***527***
⟨*ImageTexture Public Methods*⟩, 524, **527**
⟨*Incorporate results from $+z$ face to coefficients*⟩, 946, **948**
⟨*InfiniteAreaCube Public Methods*⟩, **948**, 948, ***949***
⟨*InfiniteAreaLight Declarations*⟩, **629**
⟨*InfiniteAreaLight Method Definitions*⟩, **630**, ***631***, ***727***, ***729***, ***944***
⟨*InfiniteAreaLight Private Data*⟩, 629, **631**, ***727***
⟨*InfiniteAreaLight Public Methods*⟩, 629, **631**
⟨*InfiniteAreaLight Utility Classes*⟩, **948**
⟨*Init* `SubsurfaceOctreeNode` *leaf from* `IrradiancePoint`*s*⟩, **901**, 901
⟨*Initialize* `A` *and* `C` *matrices for barycentrics*⟩, **147**, 147
⟨*Initialize* `A`, `Bx`, *and* `By` *matrices for offset computation*⟩, **508**, 508
⟨*Initialize basic variables for camera path vertex*⟩, **847**, 847, 849
⟨*Initialize* `BucketInfo` *for SAH partition buckets*⟩, 219, **220**
⟨*Initialize* `buildData` *array for primitives*⟩, **210**, 210
⟨*Initialize common variables for* `CreateRadProbeTask::Run()`⟩, **963**, 963
⟨*Initialize common variables for Whitted integrator*⟩, **43**, 43
⟨*Initialize depth of field parameters*⟩, 305, **314**
⟨*Initialize* `DifferentialGeometry` *from parameters*⟩, **102**, 102, ***505***
⟨*Initialize* `DifferentialGeometry` *from parametric information*⟩, 117, **122**, 127, 131
⟨*Initialize edges for* `axis`⟩, **236**, 236
⟨*Initialize EWA filter weights if needed*⟩, 530, **544**
⟨*Initialize first three columns of viewing matrix*⟩, 84, **85**
⟨*Initialize fourth column of viewing matrix*⟩, **84**, 84
⟨*Initialize Global Variables*⟩, **2**, 2, ***3***
⟨*Initialize interior node and continue recursion*⟩, 233, **234**
⟨*Initialize ith MIPMap level from $i-1$st level*⟩, 534, **536**
⟨*Initialize leaf node if termination criteria met*⟩, **233**, 233
⟨*Initialize levels of MIPMap from image*⟩, 530, **534**
⟨*Initialize light and bsdf samples for single light sample*⟩, 746, **747**
⟨*Initialize matrices for chosen projection plane*⟩, 508, **509**
⟨*Initialize most detailed level of MIPMap*⟩, 534, **535**
⟨*Initialize* `primitives` *with primitives for grid*⟩, 197, **198**
⟨*Initialize* `primNums` *for kd-tree construction*⟩, **232**, 232
⟨*Initialize* `ProjectionLight` *projection matrix*⟩, **616**, 616
⟨*Initialize* `samples` *with computed sample values*⟩, 375, **376**
⟨*Initialize sampling PDFs for infinite area light*⟩, 630, **725**
⟨*Initialize stratified* `samples` *with sample values*⟩, 352, **354**
⟨*Initialize* `Triangle` *shading geometry with* `n` *and* `s`⟩, 145, **146**
⟨*Initialize* `volumeRegion` *from volume region(s)*⟩, **1072**, 1072
⟨*Integrator Declarations*⟩, **740**, ***740***
⟨*Integrator Interface*⟩, **740**, ***740***, 740
⟨*Integrator Utility Functions*⟩, **46**, ***709***, ***745***, ***746***, ***749***
⟨*Interpolate (u, v) triangle parametric coordinates*⟩, 141, **143**
⟨*Interpolate rotation at* `dt`⟩, **99**, 99
⟨*Interpolate scale at* `dt`⟩, **99**, 99
⟨*Interpolate translation at* `dt`⟩, **99**, 99
⟨*Intersect ray with primitives in leaf BVH node*⟩, 225, **226**
⟨*Intersect sample ray with area light geometry*⟩, **717**, 717
⟨*Intersection Declarations*⟩, **186**
⟨*Intersection Method Definitions*⟩, **484**, ***625***
⟨*Intersection Public Data*⟩, **186**, 186
⟨*Intersection Public Methods*⟩, 186, **484**
⟨*IrradianceCacheIntegrator Declarations*⟩, **786**
⟨*IrradianceCacheIntegrator Local Declarations*⟩, **792**, ***793***
⟨*IrradianceCacheIntegrator Method Definitions*⟩, **788**, ***788***, ***790***, ***792***, ***793***
⟨*IrradianceCacheIntegrator Private Data*⟩, 786, **787**, ***788***
⟨*IrradProcess Data*⟩, 792, **793**
⟨*IrradProcess Public Methods*⟩, 792, **794**, ***794***
⟨*IrregIsoProc Public Methods*⟩, **466**, ***466***, 466
⟨*IrregIsotropicBRDF Private Data*⟩, 464, **465**
⟨*IrregIsotropicBRDF Public Methods*⟩, 464, **465**
⟨*KdAccelNode Methods*⟩, 229, **231**, ***231***
⟨*KdNode Data*⟩, 1028, **1029**
⟨*KdTree Declarations*⟩, **1028**, ***1029***, ***1031***
⟨*KdTree Method Definitions*⟩, **1029**, ***1030***, ***1032***
⟨*KdTree Private Data*⟩, **1029**, 1029
⟨*KdTreeAccel Declarations*⟩, **228**, ***241***
⟨*KdTreeAccel Local Declarations*⟩, **229**, ***235***
⟨*KdTreeAccel Method Definitions*⟩, **229**, ***230***, ***233***, ***240***
⟨*KdTreeAccel Private Data*⟩, 228, **229**, ***232***, ***234***
⟨*KdTreeAccel Public Methods*⟩, 228, **245**
⟨*Lambertian Private Data*⟩, **447**, 447
⟨*Lambertian Public Methods*⟩, **447**, ***447***, 447

⟨*LanczosSincFilter Public Methods*⟩, 400, **402**, ***402***
⟨*LDSampler Declarations*⟩, **373**
⟨*LDSampler Method Definitions*⟩, **373**, ***374***
⟨*LDSampler Private Data*⟩, 373, **374**
⟨*LDSampler Public Methods*⟩, 373, **374**, ***374***, ***377***
⟨*Light Declarations*⟩, **606**, ***608***, ***623***, ***710***
⟨*Light Interface*⟩, **606**, 606, ***608***, ***711***, ***941***
⟨*Light Method Definitions*⟩, **609**, ***609***, ***631***, ***710***, ***943***
⟨*Light Protected Data*⟩, **606**, 606
⟨*Light Public Data*⟩, **606**, 606
⟨*Look for texture in texture cache*⟩, **525**, 525
⟨*Lookup nearby photons at irradiance computation point*⟩, **832**, 832
⟨*Loop over filter support and add sample to pixel arrays*⟩, 408, **409**
⟨*Loop over light path vertices and connect to camera vertex*⟩, 850, **851**
⟨*Loop over light sources and sum their SH coefficients*⟩, **949**, 949
⟨*Loop over primitives in voxel and find intersections*⟩, 206, **207**
⟨*Loop over wavelength sample segments and add contributions*⟩, 268, **269**
⟨*LoopSubdiv Declarations*⟩, **151**
⟨*LoopSubdiv Inline Functions*⟩, **158**
⟨*LoopSubdiv Local Structures*⟩, **152**, ***153***, ***155***
⟨*LoopSubdiv Macros*⟩, **153**
⟨*LoopSubdiv Method Definitions*⟩, **151**, ***160***, ***161***, ***164***, ***166***
⟨*LoopSubdiv Private Data*⟩, 151, **152**
⟨*LoopSubdiv Private Methods*⟩, 151, **163**, ***171***
⟨*main program*⟩, **20**
⟨*Make first pass through candidate points with reader lock*⟩, 892, **894**
⟨*Make second pass through points with writer lock and update octree*⟩, 892, **895**
⟨*Map square to* (r, θ)⟩, 667, **668**
⟨*Map uniform random numbers to* $[-1, 1]^2$⟩, **667**, 667
⟨*MarbleTexture Declarations*⟩, **568**
⟨*MarbleTexture Private Data*⟩, **568**, 568
⟨*MarbleTexture Public Methods*⟩, **568**, 568, ***569***
⟨*Material Declarations*⟩, **483**
⟨*Material Interface*⟩, **483**, 483, ***484***
⟨*Material Method Definitions*⟩, **495**
⟨*Matrix4x4 Declarations*⟩, **1021**
⟨*Matrix4x4 Method Definitions*⟩, **1020**, ***1021***
⟨*Matrix4x4 Public Methods*⟩, **1021**, ***1021***, 1021
⟨*MatteMaterial Declarations*⟩, **484**
⟨*MatteMaterial Method Definitions*⟩, **485**
⟨*MatteMaterial Private Data*⟩, 484, **485**
⟨*MatteMaterial Public Methods*⟩, 484, **485**
⟨*MeasuredMaterial Declarations*⟩, **489**
⟨*MeasuredMaterial Method Definitions*⟩, **489**
⟨*MeasuredMaterial Private Data*⟩, **489**, 489
⟨*Memory Allocation Functions*⟩, **1013**
⟨*Memory Declarations*⟩, **1010**, ***1011***, ***1013***, ***1015***, ***1017***
⟨*MemoryArena Private Data*⟩, **1015**, 1015
⟨*MemoryArena Public Methods*⟩, **1015**, 1015, ***1016***, ***1017***
⟨*Merge indirect photons into shared array*⟩, **812**, 812
⟨*Merge local photon data with data in* `PhotonIntegrator`⟩, 806, **812**
⟨*Metropolis Declarations*⟩, **852**
⟨*Metropolis Local Declarations*⟩, **838**, ***838***, ***839***, ***840***, ***841***, ***842***, ***843***
⟨*Metropolis Method Definitions*⟩, **843**, ***845***, ***847***, ***849***, ***855***, ***856***, ***857***
⟨*MetropolisRenderer Private Data*⟩, 852, **853**
⟨*Microfacet Private Data*⟩, **454**, 454
⟨*Microfacet Public Methods*⟩, 454, **455**
⟨*MicrofacetDistribution Interface*⟩, **454**, 454, ***695***
⟨*MIPMap Declarations*⟩, **530**, ***530***
⟨*MIPMap Method Definitions*⟩, **530**, ***535***, ***536***, ***539***, ***540***, ***541***
⟨*MIPMap Private Data*⟩, **530**, 530, ***532***, ***535***, ***544***
⟨*MIPMap Private Methods*⟩, 530, **532**
⟨*MIPMap Public Methods*⟩, 530, **535**
⟨*Mitchell Filter Declarations*⟩, **398**
⟨*Mitchell Filter Method Definitions*⟩, **399**
⟨*MitchellFilter Public Methods*⟩, **398**, 398, ***400***
⟨*MixMaterial Declarations*⟩, **488**
⟨*MixMaterial Method Definitions*⟩, **488**
⟨*MixMaterial Private Data*⟩, **488**, 488
⟨*MixMaterial Public Methods*⟩, **488**, 488
⟨*MixTexture Declarations*⟩, **521**
⟨*MixTexture Public Methods*⟩, **521**, 521, ***522***
⟨*Modify ray for depth of field*⟩, 309, 312, **315**
⟨*Monte Carlo Function Definitions*⟩, **367**, ***652***, ***664***, ***666***, ***667***, ***671***, ***672***, ***713***, ***731***
⟨*Monte Carlo Inline Functions*⟩, **693**, ***693***
⟨*Monte Carlo Utility Declarations*⟩, **354**, ***362***, ***366***, ***367***, ***648***, ***669***, ***670***, ***672***
⟨*Normal Public Methods*⟩, 65, **66**
⟨*Normalize filter weights for texel resampling*⟩, **533**, 533
⟨*Normalize pixel with weight sum*⟩, 412, **413**
⟨*Object Creation Function Definitions*⟩, **1066**
⟨*Octree Declarations*⟩, **1023**, ***1023***, ***1026***
⟨*Octree Method Definitions*⟩, **1024**, ***1027***
⟨*Octree Private Data*⟩, **1023**, 1023
⟨*Octree Public Methods*⟩, **1023**, 1023, ***1024***, ***1027***
⟨*OrenNayar Private Data*⟩, **451**
⟨*OrenNayar Public Methods*⟩, **449**
⟨*Orient shading normal to match geometric normal*⟩, 495, **496**
⟨*OrthoCamera Private Data*⟩, 306, **307**
⟨*OrthographicCamera Declarations*⟩, **306**
⟨*OrthographicCamera Definitions*⟩, **306**, ***309***, ***310***
⟨*Otherwise add data item to octree children*⟩, **1024**, 1024
⟨*Otherwise compute* M_o *from points in leaf or recursively visit children*⟩, **911**, 911
⟨*Parallel Declarations*⟩, **1036**, ***1036***, ***1037***, ***1038***, ***1039***, ***1041***
⟨*ParamSet Declarations*⟩, **1047**, ***1048***
⟨*ParamSet Methods*⟩, **1049**, ***1050***
⟨*ParamSet Private Data*⟩, **1047**, 1047
⟨*ParamSet Public Methods*⟩, 1047, **1049**, ***1049***, ***1050***, ***1051***
⟨*ParamSetItem Data*⟩, **1048**, 1048
⟨*ParamSetItem Methods*⟩, **1048**
⟨*Parse scene from input files*⟩, **21**, 21
⟨*Parse scene from standard input*⟩, **21**, 21
⟨*Partition primitives into equally-sized subsets*⟩, **217**, 219
⟨*Partition primitives into two sets and build children*⟩, 213, **215**
⟨*Partition primitives through node's midpoint*⟩, **215**
⟨*Partition primitives using approximate SAH*⟩, **219**
⟨*PathIntegrator Declarations*⟩, **766**
⟨*PathIntegrator Method Definitions*⟩, **766**, ***767***
⟨*PathIntegrator Private Data*⟩, **766**, ***766***, 766
⟨*PathIntegrator Public Methods*⟩, **766**, 766
⟨*Perform projective divide*⟩, **311**, 311
⟨*Perform trilinear interpolation at appropriate MIPMap level*⟩, 536, **537**

⟨*Perlin Noise Data*⟩, **557**
⟨*Permute LHS samples in each dimension*⟩, **356**, 356
⟨*PermutedHalton Private Data*⟩, 367, **368**
⟨*PermutedHalton Public Methods*⟩, 367, **368**
⟨*PerspectiveCamera Declarations*⟩, **310**
⟨*PerspectiveCamera Method Definitions*⟩, **310**, ***312***
⟨*PerspectiveCamera Private Data*⟩, 310, **312**
⟨*PerspectiveCamera Public Methods*⟩, 310, **312**
⟨*PhotonIntegrator Local Declarations*⟩, **805**, ***815***, ***821***, ***822***, ***825***
⟨*PhotonIntegrator Local Definitions*⟩, **822**, ***830***, ***831***, ***832***
⟨*PhotonIntegrator Method Definitions*⟩, **803**, ***804***, ***806***, ***816***
⟨*PhotonIntegrator Private Data*⟩, **803**, ***803***, ***812***, ***813***, ***815***
⟨*PlanarMapping2D Public Methods*⟩, **518**, 518
⟨*PlasticMaterial Declarations*⟩, **487**
⟨*PlasticMaterial Method Definitions*⟩, **487**
⟨*PlasticMaterial Private Data*⟩, **487**, 487
⟨*PlasticMaterial Public Methods*⟩, **487**, 487
⟨*Point Public Data*⟩, **63**, 63
⟨*Point Public Methods*⟩, 63, **64**, ***64***
⟨*Point sample* `Checkerboard2DTexture`⟩, **548**, 548, 550
⟨*PointLight Declarations*⟩, **610**
⟨*PointLight Method Definitions*⟩, **610**, ***611***, ***712***, ***942***
⟨*PointLight Private Data*⟩, 610, **611**
⟨*PointLight Public Methods*⟩, 610, **611**
⟨*Possibly add data item to current octree node*⟩, **1024**, 1024
⟨*Possibly add emitted light at path vertex*⟩, 767, **768**
⟨*Possibly create area light for shape*⟩, 1066, **1067**
⟨*Possibly create radiance photon at photon intersection point*⟩, 810, **814**
⟨*Possibly skip virtual light shadow ray with Russian roulette*⟩, 780, **782**
⟨*Possibly terminate photon path with Russian roulette*⟩, 810, **811**
⟨*Possibly terminate ray marching if transmittance is small*⟩, 881, **882**
⟨*Possibly terminate the path*⟩, 767, **769**
⟨*Possibly terminate virtual light path with Russian roulette*⟩, 779, **780**
⟨*Precompute directions ω and SH values for directions*⟩, **978**, 978
⟨*Precompute filter weight table*⟩, 405, **407**
⟨*Precompute information for light sampling densities*⟩, 776, **777**
⟨*Precompute radiance at a subset of the photons*⟩, 804, **815**
⟨*Precompute sine and cosine terms for z-axis SH rotation*⟩, **955**, 955
⟨*Precompute x and y filter table offsets*⟩, 409, **410**
⟨*Prepare for next level of subdivision*⟩, 161, **170**
⟨*Prepare for volume integration stepping*⟩, 880, **881**
⟨*Prepare temporary array pointers for low-discrepancy camera samples*⟩, **375**, 375
⟨*Prepare temporary array pointers for low-discrepancy integrator samples*⟩, **375**, 375
⟨*Prepare to traverse kd-tree for ray*⟩, 240, **241**
⟨*Primitive Declarations*⟩, **185**
⟨*Primitive Interface*⟩, **185**, ***185***, 185, ***186***, ***187***
⟨*Primitive Method Definitions*⟩, **185**, ***186***
⟨*Primitive Protected Data*⟩, **185**, 185
⟨*Primitive Public Data*⟩, **185**, 185
⟨*Process kd-tree interior node*⟩, **242**, 242
⟨*Process kd-tree node's children*⟩, 1032, **1033**
⟨*Process scene description*⟩, 20, **21**
⟨*ProgressReporter Public Methods*⟩, **1006**, ***1006***
⟨*Project diffuse transfer function at point to SH*⟩, **972**, 972
⟨*Project* `InfiniteAreaLight` *to SH from cube map sampling*⟩, 944, **949**
⟨*Project* `InfiniteAreaLight` *to SH from lat-long representation*⟩, 944, **945**
⟨*Project* `InfiniteAreaLight` *to SH using Monte Carlo if visibility needed*⟩, **944**, 944
⟨*Project point light source to SH*⟩, **942**, 942
⟨*Project point onto projection plane and compute light*⟩, 617, **618**
⟨*ProjectionLight Declarations*⟩, **614**
⟨*ProjectionLight Method Definitions*⟩, **616**, ***617***, ***618***
⟨*ProjectionLight Private Data*⟩, 614, **616**, ***617***
⟨*ProjectiveCamera Protected Data*⟩, **305**, 305, ***306***, ***314***
⟨*Push vertices to limit surface*⟩, 161, **171**
⟨*Put far BVH node on* `todo` *stack, advance to near node*⟩, 225, **227**
⟨*Put* `vert` *one-ring in* `Pring`⟩, **164**, 164, 166
⟨*Quaternion Inline Functions*⟩, **93**, ***93***
⟨*Quaternion Method Definitions*⟩, **96**
⟨*Quaternion Public Data*⟩, **93**
⟨*Quaternion Public Methods*⟩, **92**, ***93***, ***94***
⟨*RadiancePhotonProcess Methods*⟩, **825**, 825
⟨*Randomly accept proposed path mutation (or not)*⟩, 857, **861**
⟨*Randomly choose a single light to sample,* `light`⟩, 746, **747**
⟨*Ray Public Data*⟩, 66, **67**, ***67***
⟨*Ray Public Methods*⟩, 66, **68**, ***68***
⟨*RayDifferential Public Data*⟩, **69**, 69
⟨*RayDifferential Public Methods*⟩, **69**, ***69***, 69, ***70***
⟨*Recheck candidate point and possibly add to octree*⟩, 895, **896**
⟨*Record information for current path vertex*⟩, 843, **844**
⟨*Recursively build BVH tree for primitives*⟩, 210, **211**
⟨*Recursively initialize children nodes*⟩, 234, **240**
⟨*Recursively visit children nodes to compute M_o*⟩, **911**, 911
⟨*Reference Public Methods*⟩, **1011**, ***1011***, 1011, ***1012***
⟨*Refine animated shape and create BVH if more than one shape created*⟩, 1067, **1068**
⟨*Refine instance* `Primitive`s *and create aggregate*⟩, **1070**, 1070
⟨*Refine last primitive in todo list*⟩, 186, **187**
⟨*Refine primitive* `prim` *if it's not intersectable*⟩, 206, **207**
⟨*Refine primitives in voxel if needed*⟩, **206**, 206
⟨*Reflection Declarations*⟩, **465**, ***905***
⟨*RegularHalfangleBRDF Private Data*⟩, 467, **468**
⟨*RegularHalfangleBRDF Public Methods*⟩, 467, **468**
⟨*Release temporary image memory*⟩, 412, **413**
⟨*Remove most distant photon from heap and add new photon*⟩, 822, **823**
⟨*Renderer Declarations*⟩, **24**
⟨*Renderer Interface*⟩, **24**, 24, ***25***
⟨*RenderOptions Constructor Implementation*⟩, 1056, **1057**, ***1058***, ***1069***
⟨*RenderOptions Public Data*⟩, 1056, **1057**, ***1057***, ***1058***, ***1065***, ***1068***, ***1069***

⟨*RenderOptions Public Methods*⟩, 1056, **1072**
⟨*Report any inconsistencies between intersections*⟩, 247, **249**
⟨*Report sample results to* `Sampler`*, add contributions to image*⟩, 32, **33**
⟨*Request samples for final gathering*⟩, **803**, 803
⟨*Resample image in s direction*⟩, 531, **532**
⟨*Resample image to power-of-two resolution*⟩, 530, **531**
⟨*Return* `insideDot` *result if point is inside dot*⟩, 559, **560**
⟨*Return radiance value for infinite light direction*⟩, 727, **729**
⟨*Return uniform weight if point inside sphere*⟩, **722**, 722
⟨*Return* $x \in [0, 1)$ *corresponding to sample*⟩, **649**, 649
⟨*Reverse bits of* `n`⟩, **372**, 372
⟨*RGBSpectrum Public Methods*⟩, **279**, *279*, 279, ***280***
⟨*Rotate coefficients for band* `l` *about z*⟩, **955**, 955
⟨*Rotate incident SH lighting to local coordinate frame*⟩, **981**, 981
⟨*Round up* `sz` *to minimum machine alignment*⟩, **1016**, 1016
⟨*Safely update* `Lxyz` *and* `weightSum` *even with concurrency*⟩, 410, **411**
⟨*Sample BSDF to get new path direction*⟩, 767, **768**
⟨*Sample BSDF with multiple importance sampling*⟩, 749, **750**
⟨*Sample chosen* `BxDF`⟩, 706, **707**
⟨*Sample direction for irradiance estimate ray*⟩, 795, **796**
⟨*Sample direction for outgoing Metropolis path direction*⟩, 843, **844**
⟨*Sample from first quadrant and remap to hemisphere to sample* ω_h⟩, **699**, 699
⟨*Sample gather ray direction from* `photonNum`⟩, **826**, 826
⟨*Sample illumination from lights to find path contribution*⟩, 767, **768**
⟨*Sample* `i`*th direction and compute estimate for transfer coefficients*⟩, **973**, 973
⟨*Sample light ray and apply bidirectional path tracing*⟩, 845, **846**
⟨*Sample light source with multiple importance sampling*⟩, **749**, 749
⟨*Sample Method Definitions*⟩, **344**, ***345***
⟨*Sample new photon ray direction*⟩, 808, **810**
⟨*Sample new ray direction and update weight for virtual light path*⟩, 778, **779**
⟨*Sample point on lens*⟩, 315, **317**
⟨*Sample Public Data*⟩, 343, **344**
⟨*Sample Public Methods*⟩, 343, **344**, ***344***, ***346***
⟨*Sample random direction from BSDF for final gather ray*⟩, **824**, 824
⟨*Sample random direction using photons for final gather ray*⟩, **826**, 826
⟨*Sample ray leaving light source for virtual light path*⟩, 777, **778**
⟨*Sample sphere uniformly inside subtended cone*⟩, 720, **722**
⟨*Sample uniformly on sphere if* `p` *is inside it*⟩, 720, **721**
⟨*SampledSpectrum Private Data*⟩, 266, **271**, *277*
⟨*SampledSpectrum Public Methods*⟩, **266**, 266, ***267***, ***271***, ***272***, ***273***, ***275***, ***278***
⟨*Sampler Interface*⟩, **340**, 340, ***341***, ***344***
⟨*Sampler Method Definitions*⟩, **340**, ***341***
⟨*Sampler Public Data*⟩, **340**, 340
⟨*SamplerRenderer Declarations*⟩, **25**
⟨*SamplerRenderer Method Definitions*⟩, **26**, ***27***, ***34***, ***35***
⟨*SamplerRenderer Private Data*⟩, **25**, 25, ***26***
⟨*SamplerRendererTask Declarations*⟩, **29**
⟨*SamplerRendererTask Definitions*⟩, **30**
⟨*SamplerRendererTask Private Data*⟩, 29, **30**
⟨*SamplerRendererTask Public Methods*⟩, 29, **30**
⟨*Sampling Declarations*⟩, **340**, ***342***, ***343***
⟨*Sampling Function Definitions*⟩, **352**, ***352***, ***356***, ***374***, ***375***
⟨*Sampling Inline Functions*⟩, **372**, ***372***, ***377***
⟨*Sampling Local Definitions*⟩, **368**
⟨*Scale to canonical viewing volume*⟩, 311, **312**
⟨*ScaledBxDF Public Methods*⟩, 431, **432**
⟨*ScaleTexture Declarations*⟩, **520**
⟨*ScaleTexture Public Methods*⟩, 520, **521**, ***521***
⟨*Scan over ellipse bound and compute quadratic equation*⟩, 541, **543**
⟨*Scene Constructor Implementation*⟩, **24**
⟨*Scene Declarations*⟩, **22**
⟨*Scene Method Definitions*⟩, **24**
⟨*Scene Public Data*⟩, 22, **23**, ***23***, ***24***
⟨*Scene Public Methods*⟩, 22, **23**, ***24***
⟨*SDEdge Comparison Function*⟩, **155**, 155
⟨*SDEdge Constructor*⟩, **155**, 155
⟨*SDFace Methods*⟩, 153, **159**, ***159***, ***168***
⟨*SDVertex Constructor*⟩, **152**, 152
⟨*See if any shape ids differ within samples*⟩, **388**, 388
⟨*See if hit point is inside disk radii and* ϕ_{max}⟩, 131, **132**
⟨*See if point is visible to any element of* `surfacePoints`⟩, **964**, 964
⟨*Select antialiasing method for* `Checkerboard2DTexture`⟩, 547, **548**
⟨*Select initial sample from bootstrap samples*⟩, 855, **857**
⟨*Set face to vertex pointers*⟩, 151, **154**
⟨*Set neighbor pointers in* `faces`⟩, 151, **156**
⟨*Set ray origin and direction for infinite light ray*⟩, 714, **715**
⟨*Set up 3D DDA for ray*⟩, 202, **204**
⟨*Shape Declarations*⟩, **108**
⟨*Shape Interface*⟩, 108, **109**, ***113***, ***716***
⟨*Shape Method Definitions*⟩, **108**, ***109***, ***110***, ***111***, ***113***, ***717***
⟨*Shape Public Data*⟩, **108**, 108, ***109***
⟨*ShapeSet Declarations*⟩, **626**
⟨*ShapeSet Method Definitions*⟩, **723**, ***724***
⟨*ShapeSet Private Data*⟩, **626**, 626, ***723***
⟨*Shift* `dgEval du` *in the u direction*⟩, **495**, 495
⟨*Shift* `dgEval dv` *in the v direction*⟩, 495, **496**
⟨*Shift stratified image samples to pixel coordinates*⟩, 352, **353**
⟨*Sinc Filter Declarations*⟩, **400**
⟨*Sinc Filter Method Definitions*⟩, **402**
⟨*SingleScatteringIntegrator Declarations*⟩, **882**
⟨*Skip point* `p` *if not indirectly visible from camera*⟩, 963, **964**
⟨*Solve quadratic equation for* `t` *values*⟩, 117, **118**, 123, 127
⟨*Sort samples if unordered, use sorted for returned spectrum*⟩, **267**, 267, 280
⟨*Spectral Data Declarations*⟩, **271**, ***276***
⟨*Spectrum Declarations*⟩, **264**, ***266***, ***279***
⟨*Spectrum Inline Functions*⟩, **265**
⟨*Spectrum Method Definitions*⟩, **268**, ***277***, ***279***, ***281***
⟨*Spectrum Utility Declarations*⟩, **266**, ***273***, ***274***, ***277***
⟨*SpecularReflection Private Data*⟩, **440**, 440
⟨*SpecularReflection Public Methods*⟩, **440**, 440, ***441***, ***703***
⟨*SpecularTransmission Private Data*⟩, **444**, 444
⟨*SpecularTransmission Public Methods*⟩, **444**, ***444***, 444, ***703***
⟨*Sphere Declarations*⟩, **115**

⟨*Sphere Method Definitions*⟩, **116**, ***116***, ***117***, ***123***, ***124***, ***720***, ***722***, ***1073***
⟨*Sphere Private Data*⟩, 115, **116**
⟨*Spherical Harmonics Declarations*⟩, **935**, ***935***, ***946***, ***950***, ***956***, ***981***
⟨*Spherical Harmonics Definitions*⟩, **936**, ***949***, ***951***, ***953***, ***955***, ***968***, ***973***, ***976***, ***978***
⟨*Spherical Harmonics Local Definitions*⟩, **936**, ***939***, ***940***, ***968***
⟨*Splat current and proposed samples to* `Film`⟩, 857, **860**
⟨*SpotLight Declarations*⟩, **612**
⟨*SpotLight Method Definitions*⟩, **612**, ***613***, ***614***, ***713***
⟨*SpotLight Private Data*⟩, 612, **613**
⟨*Start recursive construction of kd-tree*⟩, 232, **233**
⟨*Statistics Counters Probe Declarations*⟩, **1008**
⟨*Statistics Counters Probe Definitions*⟩, **1007**
⟨*Statistics Disabled Declarations*⟩, **1007**
⟨*Stop following paths if not finding new points*⟩, 892, **896**
⟨*Store candidate sample point at ray intersection if appropriate*⟩, 893, **894**
⟨*Store primitive ids for leaf node*⟩, **230**, 230
⟨*StratifiedSampler Declarations*⟩, **349**
⟨*StratifiedSampler Method Definitions*⟩, **349**, ***350***, ***351***
⟨*StratifiedSampler Private Data*⟩, **349**, 349
⟨*StratifiedSampler Public Methods*⟩, **349**, 349, ***359***
⟨*SubsurfaceMaterial Declarations*⟩, **599**
⟨*SubsurfaceMaterial Method Definitions*⟩, **600**
⟨*SubsurfaceMaterial Private Data*⟩, **599**, 599
⟨*SubsurfaceOctreeNode Methods*⟩, 898, **899**, ***899***, ***901***
⟨*SubsurfaceOctreeNode Public Data*⟩, 898, **899**
⟨*SurfaceIntegrator Interface*⟩, 740, **741**
⟨*SurfacePoint Data*⟩, 890, **891**
⟨*SurfacePointsRenderer Declarations*⟩, **890**
⟨*SurfacePointsRenderer Local Declarations*⟩, **895**
⟨*SurfacePointsRenderer Method Definitions*⟩, **890**, ***892***, ***896***
⟨*SurfacePointsRenderer Private Data*⟩, **889**, ***890***
⟨*SurfacePointsRenderer Public Methods*⟩, **888**
⟨*Take initial set of samples to compute b*⟩, **855**, 855
⟨*Terminate path with RR or prepare for finding next vertex*⟩, 843, **844**
⟨*Test cylinder intersection against clipping parameters*⟩, 127, **128**
⟨*Test disk ϕ value against $\phi_{\max}$*⟩, **132**, 132
⟨*Test intersection against alpha texture, if present*⟩, 141, **144**
⟨*Test sphere intersection against clipping parameters*⟩, 117, **120**, 123
⟨*Texture Declarations*⟩, **514**, ***515***, ***516***, ***517***, ***518***, ***519***, ***533***
⟨*Texture Inline Functions*⟩, **563**
⟨*Texture Interface*⟩, 519, **520**
⟨*Texture Method Definitions*⟩, **515**, ***515***, ***516***, ***517***, ***518***, ***519***, ***555***, ***557***, ***558***, ***561***, ***564***
⟨*TextureMapping2D Interface*⟩, **514**, 514
⟨*TextureMapping3D Interface*⟩, **519**, 519
⟨*TextureParams Declarations*⟩, **1061**
⟨*TextureParams Method Definitions*⟩, **1062**
⟨*TextureParams Private Data*⟩, **1061**, 1061
⟨*TextureParams Public Methods*⟩, **1061**, 1061, ***1062***
⟨*Trace BSDF final gather ray and accumulate radiance*⟩, **824**, 824
⟨*Trace photon-sampled final gather ray and accumulate radiance*⟩, 826, **827**
⟨*Trace ray for bias compensation gather sample*⟩, **783**, 783
⟨*Trace ray to sample radiance for irradiance estimate*⟩, 795, **796**
⟨*Trace rays for specular reflection and refraction*⟩, 43, **46**, 789
⟨*Transform Declarations*⟩, **76**
⟨*Transform Inline Functions*⟩, **86**, ***86***, ***88***
⟨*Transform instance's differential geometry to world space*⟩, 190, **191**
⟨*Transform mesh vertices to world space*⟩, 137, **138**
⟨*Transform Method Definitions*⟩, **79**, ***80***, ***81***, ***83***, ***84***, ***88***, ***89***, ***307***, ***311***
⟨*Transform Private Data*⟩, **76**, 76
⟨*Transform Public Methods*⟩, 76, **77**, ***77***, ***80***
⟨*Transform* `Ray` *to object space*⟩, **117**, 117, 123, 127, 131
⟨*Transform step function integral into CDF*⟩, 648, **649**
⟨*TransformCache Private Data*⟩, **1066**
⟨*TransformedPrimitive Declarations*⟩, **190**
⟨*TransformedPrimitive Method Definitions*⟩, **190**
⟨*TransformedPrimitive Private Data*⟩, **190**, 190
⟨*TransformedPrimitive Public Methods*⟩, **190**, 190, ***191***
⟨*TransformSet Public Methods*⟩, **1054**, ***1054***, 1054, ***1064***
⟨*Traverse kd-tree nodes in order for ray*⟩, 240, **242**
⟨*Triangle Filter Declarations*⟩, **396**
⟨*Triangle Filter Method Definitions*⟩, **396**
⟨*Triangle Private Data*⟩, **139**, 139
⟨*Triangle Public Methods*⟩, **139**, 139, ***144***
⟨*TriangleMesh Declarations*⟩, **135**, ***139***
⟨*TriangleMesh Method Definitions*⟩, **137**, ***138***, ***139***, ***140***, ***141***, ***145***, ***719***
⟨*TriangleMesh Protected Data*⟩, 135, **137**
⟨*TriangleMesh Public Methods*⟩, 135, **138**
⟨*Trilinearly interpolate density values to compute local density*⟩, 593, **594**
⟨*Try to compute radiance probe contribution at* `i`*th sample point*⟩, **963**, 963
⟨*Try to find enough BRDF samples around* `m` *within search radius*⟩, **466**, 466
⟨*Try to generate next vertex of ray path*⟩, **843**, 843
⟨*Update best split if this is lowest cost so far*⟩, **238**, 238
⟨*Update BSDF matrix elements for sampled directions*⟩, **979**, 979
⟨*Update child vertex pointer to new even vertex*⟩, **170**, 170
⟨*Update child vertex pointer to new odd vertex*⟩, **170**, 170
⟨*Update children* `f` *pointers for neighbor children*⟩, 169, **170**
⟨*Update children* `f` *pointers for siblings*⟩, **169**, 169
⟨*Update even vertex face pointers*⟩, **169**, 169
⟨*Update* `f` *and* `v` *for next level of subdivision*⟩, **161**, 161
⟨*Update face neighbor pointers*⟩, **169**, 169
⟨*Update face vertex pointers*⟩, 169, **170**
⟨*Update for rejected candidate point*⟩, **895**, 895, 896
⟨*Update interval for* `i`*th bounding box slab*⟩, 194, **195**
⟨*Update neighbor pointer for* `edgeNum`⟩, **156**, 156
⟨*Update new mesh topology*⟩, 161, **169**
⟨*Update parametric interval from slab intersection* `t`*s*⟩, **195**, 195
⟨*Update pixel values with filtered sample contribution*⟩, 409, **410**
⟨*Update ray for effect of lens*⟩, 315, **318**
⟨*Update sample shifts*⟩, **384**, 384

⟨*Update* `tHit` *for quadric intersection*⟩, 117, **123**, 127, 131
⟨*Update transfer matrix for unoccluded direction*⟩, **977**, 977
⟨*Update vertex positions and create new edge vertices*⟩, 161, **162**
⟨*Update vertex positions for even vertices*⟩, **162**, 162
⟨*Use BSDF to do final gathering*⟩, 819, **824**
⟨*Use hierarchical integration to evaluate reflection from dipole model*⟩, 908, **910**
⟨*Use* `n` *and* `s` *to compute shading tangents for triangle,* `ss` *and* `ts`⟩, 146, **147**
⟨*Use nearby photons to do final gathering*⟩, 819, **826**
⟨*UseRadianceProbes Private Data*⟩, **965**
⟨*UVTexture Declarations*⟩, **545**
⟨*UVTexture Public Methods*⟩, 545, **546**
⟨*Vector Public Data*⟩, **57**, 57
⟨*Vector Public Methods*⟩, 57, **58**, ***58***, ***59***, ***60***, ***62***, ***66***
⟨*VisibilityTester Public Methods*⟩, 608, **609**, ***609***
⟨*Volume Scattering Declarations*⟩, **585**, ***587***, ***591***, ***597***, ***600***
⟨*Volume Scattering Definitions*⟩, **584**, ***585***, ***586***, ***588***, ***597***, ***598***, ***733***, ***913***
⟨*Volume Scattering Local Definitions*⟩, **912**, ***912***
⟨*VolumeGridDensity Declarations*⟩, **593**
⟨*VolumeGridDensity Method Definitions*⟩, **593**
⟨*VolumeGridDensity Private Data*⟩, **593**, 593
⟨*VolumeGridDensity Public Methods*⟩, **593**, 593, ***594***
⟨*VolumeIntegrator Interface*⟩, **876**, ***876***
⟨*VolumeRegion Interface*⟩, **587**, 587, ***588***
⟨*Voxel Declarations*⟩, **202**
⟨*Voxel Public Methods*⟩, **202**, ***202***, 202
⟨*Walk ray through voxel grid*⟩, 202, **205**
⟨*WhittedIntegrator Declarations*⟩, **42**
⟨*WhittedIntegrator Method Definitions*⟩, **42**
⟨*WhittedIntegrator Private Data*⟩, **42**, 42
⟨*WindyTexture Declarations*⟩, **567**
⟨*WindyTexture Private Data*⟩, **567**, 567
⟨*WindyTexture Public Methods*⟩, **567**, ***567***, 567
⟨*Write RGB image*⟩, 412, **413**

This page intentionally left blank

INDEX OF CLASSES AND THEIR MEMBERS

Bold numbers indicate the page of a class definition. Class methods and fields are indented.

AdaptiveSampler, **386**
 GetMoreSamples(), 387
 GetSubSampler(), 386
 MaxmimumSampleCount(), 386
 maxSamples, 386
 method, 386
 minSamples, 386
 needsSupersampling(), 388
 ReportResults(), 387
 RoundSize(), 386
 sampleBuf, 386
 supersamplePixel, 386
 xPos, 386
 yPos, 386
Aggregate, **192**
AggregateTest, **247**
 bboxes, 247
 nIterations, 247
 primitives, 247
 Render(), 247
AggregateVolume, **597**
 bound, 597
 IntersectP(), 598
 Lve(), 597
 p(), 597
 regions, 597
 sigma_a(), 597
 sigma_s(), 597
 sigma_t(), 597
AnimatedTransform, **96**
 actuallyAnimated, 96
 Decompose(), 97
 endTime, 96
 endTransform, 96
 Interpolate(), 99
 MotionBounds(), 100
 R, 96
 S, 96
 startTime, 96
 startTransform, 96
 T, 96
Anisotropic, **458**
 D(), 460
 ex, 458
 ey, 458
 Pdf(), 700
 sampleFirstQuadrant(), 700
 Sample_f(), 699
AreaLight, **623**
 L(), 625
AtomicInt32, **1036**
AtomicInt64, **1036**
BBox, **70**
 BoundingSphere(), 74
 Expand(), 72
 Inside(), 72
 IntersectP(), 194
 Lerp(), 73
 MaximumExtent(), 73
 Offset(), 73
 operator[], 73
 Overlaps(), 72
 pMax, 71
 pMin, 71
 SurfaceArea(), 72
 Union(), 71
 Volume(), 72
BestCandidateSampler, **381**
 GetMoreSamples(), 383
 GetSubSampler(), 383
 MaximumSampleCount(), 383
 RoundSize(), 383
 sampleOffsets, 384
 sampleTable, 383
 tableOffset, 381
 tableWidth, 381
 xTile, 381
 xTileEnd, 381
 xTileStart, 381
 yTile, 381
 yTileEnd, 381
 yTileStart, 381
BilerpTexture, **522**
 Evaluate(), 523
 mapping, 522
 v00, 522
 v01, 522
 v10, 522
 v11, 522

Blinn, **456**
D(), 457
exponent, 456
Pdf(), 699
Sample_f(), 696
BlockedArray, **1017**
Block(), 1019
BlockSize(), 1019
data, 1018
logBlockSize, 1017
Offset(), 1019
operator(), 1020
RoundUp(), 1019
uBlocks, 1018
uRes, 1018
uSize(), 1019
vRes, 1018
vSize(), 1019
BoundEdge, **235**
END, 235
primNum, 235
START, 235
t, 235
type, 235
BoxFilter, **396**
Evaluate(), 396
BRDFToBTDF, **431**
brdf, 431
f(), 431
otherHemisphere(), 431
BSDF, **478**
Add(), 479
bxdfs, 479
dgShading, 479
eta, 479
f(), 481
LocalToWorld(), 480
MAX_BxDFS, 479
nBxDFs, 479
ng, 479
nn, 479
NumComponents(), 479
Pdf(), 708
rho(), 482
Sample_f(), 706
sn, 479
tn, 479
WorldToLocal(), 480
BSDFSample, **705**
uComponent, 705
uDir, 705
BSDFSampleOffsets, **706**
componentOffset, 706
dirOffset, 706
nSamples, 706
e, 599
eta(), 599
sigma_a(), 599
sigma_prime_s(), 599
sigp_s, 599
sig_a, 599
bounds, 219
count, 219
BVHAccel, **209**
flattenBVHTree(), 224
Intersect(), 224
IntersectP(), 227
maxPrimsInNode, 210
nodes, 223
primitives, 210
recursiveBuild(), 213
splitMethod, 210
BVHBuildNode, **211**
bounds, 211
children, 211
firstPrimOffset, 211
InitInterior(), 212
InitLeaf(), 212
nPrimitives, 211
splitAxis, 211
BVHPrimitiveInfo, **211**
bounds, 211
centroid, 211
primitiveNumber, 211
BxDF, **428**
f(), 429
MatchesFlags(), 429
Pdf(), 695
rho(), 430
Sample_f(), 694
type, 429
BxDFType, **428**
Camera, **302**
CameraToWorld, 302
film, 302
GenerateRay(), 303
GenerateRayDifferential(), 303
shutterClose, 302
shutterOpen, 302
CameraSample, **342**
imageX, 342
imageY, 342
lensU, 342
lensV, 342
time, 342
Checkerboard2DTexture, **547**
aaMethod, 548
CLOSEDFORM, 548
Evaluate(), 548
mapping, 547
NONE, 548
tex1, 547
tex2, 547
Checkerboard3DTexture, **552**
Evaluate(), 553
mapping, 553
tex1, 553
tex2, 553
ClosePhoton, **822**
distanceSquared, 822
photon, 822
CoefficientSpectrum, **264**
c, 264
CompareNode, **1031**
Run(), 816
Cone, **133**
ConstantTexture, **520**
Evaluate(), 520
CreateRadianceProbes, **958**
bbox, 958
camera, 958
filename, 958
includeDirectInProbes, 958
includeIndirectInProbes, 958
Li(), 958
lmax, 958
nIndirSamples, 958
probeSpacing, 958
Render(), 959
surfaceIntegrator, 958
time, 958
Transmittance(), 958
volumeIntegrator, 958
Run(), 963
Cylinder, **124**
Area(), 129
Intersect(), 127
ObjectBound(), 126
phiMax, 126
radius, 126
Sample(), 719
zmax, 126
zmin, 126
CylindricalMapping2D, **517**
cylinder(), 517
Map(), 517
WorldToTexture, 517
DensityRegion, **591**
Density(), 592
g, 592
le, 592
Lve(), 592
p(), 592
sigma_a(), 592
sigma_s(), 592
sigma_t(), 592
sig_a, 592
sig_s, 592

tau(), 733
WorldToVolume, 592
DifferentialGeometry, **102**
ComputeDifferentials(), 505
dndu, 102
dndv, 102
dpdu, 102
dpdv, 102
dpdx, 505
dpdy, 505
dudx, 505
dudy, 505
dvdx, 505
dvdy, 505
nn, 102
p, 102
shape, 102
u, 102
v, 102
DiffuseAreaLight, **625**
area, 626
IsDeltaLight(), 627
L(), 626
Lemit, 626
Pdf(), 718
Power(), 627
Sample_L(), 718
shapeSet, 626
DiffusePRTIntegrator, **972**
c_in, 972
Li(), 972
lmax, 972
nSamples, 972
Preprocess(), 972
DiffusionReflectance, **907**
operator(), 908
DipoleSubsurfaceIntegrator, **887**
filename, 888
irradiancePoints, 897
Li(), 908
maxError, 888
maxSpecularDepth, 888
minSampleDist, 888
octree, 898
octreeArena, 898
octreeBounds, 898
Preprocess(), 897
DirectLightingIntegrator, **742**
bsdfSampleOffsets, 743
Li(), 744
lightNumOffset, 742
lightSampleOffsets, 743
RequestSamples(), 743
SAMPLE_ALL_UNIFORM, 742
SAMPLE_ONE_UNIFORM, 742
strategy, 742
Disk, **129**
Area(), 132
height, 130
innerRadius, 130
Intersect(), 131
ObjectBound(), 131
phiMax, 130
radius, 130
Sample(), 719
DistantLight, **621**
L, 622
lightDir, 622
Power(), 623
Sample_L(), 714
Distribution1D, **648**
cdf, 648
count, 648
func, 648
funcInt, 648
SampleContinuous(), 649
SampleDiscrete(), 650
Distribution2D, **672**
pConditionalV, 672
Pdf(), 673
pMarginal, 673
SampleContinuous(), 673
DotsTexture, **559**
Evaluate(), 559
insideDot, 559
mapping, 559
outsideDot, 559
EmissionIntegrator, **877**
Li(), 880
RequestSamples(), 877
scatterSampleOffset, 877
stepSize, 877
tauSampleOffset, 877
Transmittance(), 879
EnvironmentCamera, **318**
GenerateRay(), 319
ExponentialDensity, **595**
a, 596
b, 596
Density(), 596
extent, 596
upDir, 596
FBmTexture, **565**
Evaluate(), 566
mapping, 566
octaves, 566
omega, 566
Film, **403**
AddSample(), 403
GetPixelExtent(), 404
GetSampleExtent(), 404
Splat(), 403
UpdateDisplay(), 404
WriteImage(), 404
xResolution, 403
yResolution, 403
Filter, **393**
Evaluate(), 394
invXWidth, 394
invYWidth, 394
xWidth, 394
yWidth, 394
Fresnel, **436**
Evaluate(), 436
FresnelBlend, **460**
distribution, 461
f(), 462
Pdf(), 701
Rd, 461
Rs, 461
Sample_f(), 701
SchlickFresnel(), 462
FresnelConductor, **436**
eta, 436
Evaluate(), 437
k, 436
FresnelDielectric, **437**
eta_i, 437
eta_t, 437
Evaluate(), 437
FresnelNoOp, **439**
Evaluate(), 439
GaussianFilter, **397**
alpha, 397
Evaluate(), 398
expX, 397
expY, 397
GeometricPrimitive, **188**
areaLight, 188
CanIntersect(), 188
GetAreaLight(), 188
GetBSDF(), 189
GetBSSRDF(), 189
Intersect(), 188
IntersectP(), 188
material, 188
Refine(), 188
shape, 188
WorldBound(), 188
GlossyPRTIntegrator, **980**
B, 980
c_in, 980
Kd, 980
Ks, 980
Li(), 980
lmax, 980
nSamples, 980
roughness, 980

GonioPhotometricLight, **618**
Intensity, 620
lightPos, 620
mipmap, 620
Power(), 621
GraphicsState, **1059**
areaLight, 1065
areaLightParams, 1065
CreateMaterial(), 1067
currentNamedMaterial, 1064
floatTextures, 1064
material, 1064
materialParams, 1064
namedMaterials, 1064
spectrumTextures, 1064
GridAccel, **196**
bounds, 198
CanIntersect(), 202
Intersect(), 202
IntersectP(), 208
invWidth, 199
nVoxels, 198
offset(), 201
posToVoxel(), 200
primitives, 198
rwMutex, 205
voxels, 199
voxelToPos(), 200
width, 199
WorldBound(), 202
HaltonSampler, **364**
currentSample, 364
GetMoreSamples(), 364
wantedSamples, 364
HomogeneousVolumeDensity, **589**
extent, 590
g, 590
IntersectP(), 590
le, 590
Lve(), 590
p(), 590
sigma_a(), 591
sigma_s(), 590
sigma_t(), 590
sig_a, 590
sig_s, 590
tau(), 591
WorldBound(), 590
WorldToVolume, 590
Hyperboloid, **134**
IdentityMapping3D, **519**
Map(), 519
WorldToTexture, 519
IGIIntegrator, **773**
gatherSampleOffset, 775
gLimit, 774
maxSpecularDepth, 774
nGatherSamples, 774
nLightPaths, 773
nLightSets, 773
Preprocess(), 776
RequestSamples(), 774
rrThreshold, 774
virtualLights, 779
vlSetOffset, 775
ImageFilm, **404**
AddSample(), 408
cropWindow, 405
filename, 405
filter, 405
filterTable, 407
GetPixelExtent(), 412
GetSampleExtent(), 411
pixels, 406
Splat(), 411
WriteImage(), 412
xPixelCount, 406
xPixelStart, 406
yPixelCount, 406
yPixelStart, 406
ImageTexture, **524**
ClearCache(), 527
convertIn(), 526
convertOut(), 527
Evaluate(), 527
GetTexture(), 525
mapping, 524
mipmap, 524
textures, 525
InfiniteAreaCube, **948**
computeVis, 948
light, 948
operator(), 949
pEpsilon, 948
scene, 948
time, 948
InfiniteAreaLight, **629**
distribution, 727
Le(), 631
Pdf(), 729
Power(), 631
radianceMap, 631
Sample_L(), 727
SHProject(), 944
Integrator, **740**
Preprocess(), 740
RequestSamples(), 740
Intersection, **186**
dg, 186
GetBSDF(), 484
GetBSSRDF(), 484
Le(), 625
ObjectToWorld, 186
primitive, 186
primitiveId, 186
rayEpsilon, 186
shapeId, 186
WorldToObject, 186
IrradianceCacheIntegrator, **786**
cosMaxSampleAngleDifference, 787
indirectLo(), 790
interpolateE(), 792
Li(), 789
maxIndirectDepth, 787
maxSamplePixelSpacing, 787
maxSpecularDepth, 787
minSamplePixelSpacing, 787
minWeight, 787
mutex, 787
nSamples, 787
octree, 788
Preprocess(), 788
RequestSamples(), 788
IrradiancePoint, **898**
area, 898
E, 898
n, 898
p, 898
rayEpsilon, 898
IrradianceSample, **793**
E, 793
maxDist, 793
n, 793
p, 793
wAvg, 793
IrradProcess, **792**
cosMaxSampleAngleDifference, 793
E, 793
GetAverageDirection(), 794
GetIrradiance(), 794
minWeight, 793
n, 793
p, 793
Successful(), 794
sumWt, 793
wAvg, 793
IrregIsoProc, **466**
nFound, 466
sumWeights, 466
v, 466
IrregIsotropicBRDF, **464**
f(), 466
isoBRDFData, 465
IrregIsotropicBRDFSample, **465**
p, 465
v, 465
KdAccelNode, **229**
aboveChild, 229

AboveChild(), 231
flags, 229
initInterior(), 231
initLeaf(), 230
IsLeaf(), 231
nPrimitives(), 231
nPrims, 229
onePrimitive, 229
primitives, 229
split, 229
SplitAxis(), 231
SplitPos(), 231
KdNode, **1028**
hasLeftChild, 1029
init(), 1028
initLeaf(), 1028
rightChild, 1029
splitAxis, 1029
splitPos, 1029
KdSubsurfaceMaterial, **600**
KdToDo, **241**
node, 241
tmax, 241
tmin, 241
KdTree, **1029**
Lookup(), 1032
nextFreeNode, 1029
nNodes, 1029
nodeData, 1029
nodes, 1029
privateLookup(), 1032
recursiveBuild(), 1030
KdTreeAccel, **228**
arena, 234
bounds, 232
buildTree(), 233
emptyBonus, 229
Intersect(), 240
IntersectP(), 245
isectCost, 229
maxDepth, 229
maxPrims, 229
nAllocedNodes, 232
nextFreeNode, 232
nodes, 232
primitives, 229
traversalCost, 229
Lambertian, **447**
f(), 447
R, 447
rho(), 447
LanczosSincFilter, **400**
Evaluate(), 402
Sinc1D(), 402
tau, 400
LDSampler, **373**
GetMoreSamples(), 374
GetSubSampler(), 374
MaximumSampleCount(), 377
nPixelSamples, 374
RoundSize(), 374
sampleBuf, 374
xPos, 374
yPos, 374
Light, **606**
IsDeltaLight(), 608
Le(), 631
LightToWorld, 606
nSamples, 606
Pdf(), 711
Power(), 608
Sample_L(), 608
SHProject(), 943
WorldToLight, 606
LightingSample, **838**
bsdfSample, 838
lightNum, 838
lightSample, 838
LightSample, **710**
uComponent, 710
uPos, 710
LightSampleOffsets, **710**
componentOffset, 710
nSamples, 710
posOffset, 710
LinearBVHNode, **222**
axis, 222
bounds, 222
nPrimitives, 222
primitivesOffset, 222
secondChildOffset, 222
LoopSubdiv, **151**
beta(), 171
CanIntersect(), 160
faces, 152
gamma(), 171
nLevels, 152
Refine(), 161
vertices, 152
weightBoundary(), 166
weightOneRing(), 164
MarbleTexture, **568**
Evaluate(), 569
mapping, 568
octaves, 568
omega, 568
scale, 568
variation, 568
Material, **483**
Bump(), 495
GetBSDF(), 483
GetBSSRDF(), 484
Matrix4x4, **1021**
m, 1021
Mul(), 1021
MatteMaterial, **484**
bumpMap, 485
GetBSDF(), 485
Kd, 485
sigma, 485
MeasuredMaterial, **489**
bumpMap, 489
GetBSDF(), 489
nPhiD, 489
nThetaD, 489
nThetaH, 489
regularHalfangleData, 489
thetaPhiData, 489
MemoryArena, **1015**
Alloc(), 1016
availableBlocks, 1015
blockSize, 1015
curBlockPos, 1015
currentBlock, 1015
FreeAll(), 1017
usedBlocks, 1015
MetropolisRenderer, **852**
bidirectional, 853
camera, 853
directLighting, 853
largeStepsPerPixel, 853
Lbidir(), 849
Lpath(), 847
maxConsecutiveRejects, 853
maxDepth, 853
nBootstrap, 853
nDirectPixelSamples, 853
nPixelSamples, 853
PathL(), 845
Render(), 855
Microfacet, **454**
distribution, 454
f(), 455
fresnel, 454
G(), 455
Pdf(), 696
R, 454
Sample_f(), 696
MicrofacetDistribution, **454**
D(), 454
Pdf(), 695
Sample_f(), 695
MIPMap, **530**
doTrilinear, 530
EWA(), 541
height, 535
Height(), 535

MIPMap *(continued)*
Levels(), 535
Lookup(), 540
Lookup(tri), 536
maxAnisotropy, 530
nLevels, 535
pyramid, 535
resampleWeights(), 532
Texel(), 535
triangle(), 539
weightLut, 544
WEIGHT_LUT_SIZE, 544
width, 535
Width(), 535
wrapMode, 530
MitchellFilter, **398**
B, 398
C, 398
Evaluate(), 399
Mitchell1D(), 400
MixMaterial, **488**
GetBSDF(), 488
m1, 488
m2, 488
scale, 488
MixTexture, **521**
amount, 521
Evaluate(), 522
tex1, 521
tex2, 521
MLTSample, **839**
cameraPathSamples, 839
cameraSample, 839
lightingSamples, 839
lightNumSample, 839
lightPathSamples, 839
lightRaySamples, 839
dx, 859
dy, 859
Run(), 857
Mutex, **1038**
Create(), 1038
Destroy(), 1038
MutexLock, **1039**
NodeData, **1023**
Normal, **65**
Normalize(), 65
OctNode, **1023**
children, 1023
data, 1023
Octree, **1023**
Add(), 1024
addPrivate(), 1024
bound, 1023
Lookup(), 1027
lookupPrivate(), 1027
maxDepth, 1023
root, 1023
Options, **1051**
imageFile, 1051
nCores, 1051
openWindow, 1051
quickRender, 1051
quiet, 1051
verbose, 1051
OrenNayar, **449**
A, 451
B, 451
f(), 451
R, 451
OrthoCamera, **306**
dxCamera, 307
dyCamera, 307
GenerateRay(), 309
GenerateRayDifferential(), 310
Paraboloid, **134**
ParamSet, **1047**
AddBlackbodySpectrum(), 1049
AddBool(), 1049
AddFloat(), 1049
AddInt(), 1049
AddNormal(), 1049
AddPoint(), 1049
AddRGBSpectrum(), 1049
AddSampledSpectrum(), 1049
AddSampledSpectrumFiles(), 1049
AddString(), 1049
AddTexture(), 1049
AddVector(), 1049
AddXYZSpectrum(), 1049
bools, 1047
Clear(), 1051
FindBool(), 1050
FindFloat(), 1050
FindInt(), 1050
FindNormal(), 1050
FindOneBool(), 1050
FindOneFloat(), 1050
FindOneInt(), 1050
FindOneNormal(), 1050
FindOnePoint(), 1050
FindOneSpectrum(), 1050
FindOneString(), 1050
FindOneVector(), 1050
FindPoint(), 1050
FindSpectrum(), 1050
FindString(), 1050
FindTexture(), 1050
FindVector(), 1050
floats, 1047
ints, 1047
normals, 1047
points, 1047
ReportUnused(), 1051
spectra, 1047
strings, 1047
textures, 1047
vectors, 1047
ParamSetItem, **1048**
data, 1048
lookedUp, 1048
name, 1048
nItems, 1048
PathIntegrator, **766**
bsdfSampleOffsets, 766
Li(), 767
lightNumOffset, 766
lightSampleOffsets, 766
maxDepth, 766
pathSampleOffsets, 766
RequestSamples(), 766
SAMPLE_DEPTH, 766
PathSample, **838**
bsdfSample, 838
rrSample, 838
PathVertex, **843**
alpha, 843
bsdf, 843
isect, 843
nSpecularComponents, 843
specularBounce, 843
wNext, 843
wPrev, 843
PbrtOptions, **1051**
PermutedHalton, **367**
b, 368
dims, 368
permute, 368
Sample(), 368
PerspectiveCamera, **310**
dxCamera, 312
dyCamera, 312
GenerateRay(), 312
GenerateRayDifferential(), 312
Photon, **805**
alpha, 805
p, 805
wi, 805
PhotonIntegrator, **802**
bsdfGatherSampleOffsets, 803
causticMap, 813
cosGatherAngle, 803
finalGather, 803
gatherSamples, 803
indirectMap, 813
indirGatherSampleOffsets, 803
Li(), 817
maxDistSquared, 803

maxPhotonDepth, 803
maxSpecularDepth, 803
nCausticPaths, 812
nCausticPhotonsWanted, 803
nIndirectPaths, 812
nIndirectPhotonsWanted, 803
nLookup, 803
Preprocess(), 804
radianceMap, 815
RequestSamples(), 803
PhotonProcess, **821**
nFound, 821
nLookup, 821
operator(), 822
photons, 821
PhotonShootingTask, **805**
Run(), 806
Pixel, **406**
Lxyz, 406
splatXYZ, 406
weightSum, 406
PlanarMapping2D, **518**
ds, 518
dt, 518
map, 518
vs, 518
vt, 518
PlasticMaterial, **487**
bumpMap, 487
GetBSDF(), 487
Kd, 487
Ks, 487
roughness, 487
Point, **63**
PointLight, **610**
Intensity, 611
IsDeltaLight(), 611
lightPos, 611
Pdf(), 712
Power(), 611
Sample_L(), 611
SHProject(), 942
PoissonCheck, **895**
Primitive, **185**
CanIntersect(), 186
FullyRefine(), 186
GetAreaLight(), 187
GetBSDF(), 187
GetBSSRDF(), 187
Intersect(), 186
IntersectP(), 186
nextprimitiveId, 185
primitiveId, 185
Refine(), 186
WorldBound(), 185
ProgressReporter, **1006**
Done(), 1006
Update(), 1006
ProjectionLight, **614**
cosTotalWidth, 617
hither, 617
Intensity, 616
lightPos, 616
lightProjection, 617
Power(), 618
Projection(), 617
projectionMap, 616
Sample_L(), 617
screenX0, 617
screenX1, 617
screenY0, 617
screenY1, 617
yon, 617
ProjectiveCamera, **305**
CameraToScreen, 305
focalDistance, 314
lensRadius, 314
RasterToCamera, 305
RasterToScreen, 306
ScreenToRaster, 306
Quaternion, **92**
Dot(), 93
Normalize(), 93
ToTransform(), 94
v, 93
w, 93
RadiancePhoton, **815**
Lo, 815
n, 815
p, 815
RadiancePhotonProcess, **825**
n, 825
operator(), 825
photon, 825
Ray, **66**
d, 67
depth, 67
maxt, 67
mint, 67
o, 67
operator(), 68
time, 67
RayDifferential, **69**
hasDifferentials, 69
rxDirection, 69
rxOrigin, 69
ryDirection, 69
ryOrigin, 69
ScaleDifferentials(), 70
Reference, **1011**
GetPtr(), 1012
ptr, 1011
ReferenceCounted, **1010**
nReferences, 1010
RegularHalfangleBRDF, **467**
brdf, 468
f(), 469
nPhiD, 468
nThetaD, 468
nThetaH, 468
Renderer, **24**
Li(), 25
Render(), 24
Transmittance(), 25
RenderOptions, **1056**
CameraName, 1058
CameraParams, 1058
CameraToWorld, 1058
currentInstance, 1069
FilterName, 1057
FilterParams, 1057
instances, 1069
lights, 1065
MakeRenderer(), 1072
MakeScene(), 1072
primitives, 1068
transformEndTime, 1057
transformStartTime, 1057
ResampleWeight, **532**
firstTexel, 532
weight, 532
RGB2SpectLambda, **276**
RGBIllum2SpectBlue, **276**
RGBIllum2SpectCyan, **276**
RGBIllum2SpectGreen, **276**
RGBIllum2SpectMagenta, **276**
RGBIllum2SpectRed, **276**
RGBIllum2SpectWhite, **276**
RGBIllum2SpectYellow, **276**
RGBRefl2SpectBlue, **276**
RGBRefl2SpectCyan, **276**
RGBRefl2SpectGreen, **276**
RGBRefl2SpectMagenta, **276**
RGBRefl2SpectRed, **276**
RGBRefl2SpectWhite, **276**
RGBRefl2SpectYellow, **276**
RGBSpectrum, **279**
FromRGB(), 279
FromSampled(), 280
FromXYZ(), 280
ToRGB(), 280
ToRGBSpectrum(), 280
ToXYZ(), 280
y(), 280
RandomFloat(), 1003
RandomUInt(), 1003
Seed(), 1003

RWMutex, **1039**
 Create(), 1039
 Destroy(), 1039
RWMutexLock, **1039**
 DowngradeToRead(), 1039
 UpgradeToWrite(), 1039
RWMutexLockType, **1039**
Sample, **343**
 Add1D(), 344
 Add2D(), 344
 AllocateSampleMemory(), 345
 Duplicate(), 346
 n1D, 344
 n2D, 344
 oneD, 344
 twoD, 344
SampledSpectrum, **266**
 FromRGB(), 277
 FromSampled(), 267
 FromXYZ(), 278
 Init(), 271
 rgbIllum2SpectBlue, 277
 rgbIllum2SpectCyan, 277
 rgbIllum2SpectGreen, 277
 rgbIllum2SpectMagenta, 277
 rgbIllum2SpectRed, 277
 rgbIllum2SpectWhite, 277
 rgbIllum2SpectYellow, 277
 rgbRefl2SpectBlue, 277
 rgbRefl2SpectCyan, 277
 rgbRefl2SpectGreen, 277
 rgbRefl2SpectMagenta, 277
 rgbRefl2SpectRed, 277
 rgbRefl2SpectWhite, 277
 rgbRefl2SpectYellow, 277
 ToRGB(), 275
 ToRGBSpectrum(), 275
 ToXYZ(), 272
 X, 271
 Y, 271
 yint, 271
 Z, 271
Sampler, **340**
 ComputeSubWindow(), 341
 GetMoreSamples(), 340
 GetSubSampler(), 341
 MaximumSampleCount(), 341
 ReportResults(), 341
 RoundSize(), 344
 samplesPerPixel, 340
 shutterClose, 340
 shutterOpen, 340
 xPixelEnd, 340
 xPixelStart, 340
 yPixelEnd, 340
 yPixelStart, 340
SamplerRenderer, **25**
 camera, 26
 Li(), 34
 Render(), 27
 sampler, 25
 surfaceIntegrator, 26
 Transmittance(), 35
 volumeIntegrator, 26
SamplerRendererTask, **29**
 camera, 30
 mainSampler, 30
 origSample, 30
 renderer, 30
 Run(), 30
 scene, 30
 taskCount, 30
 taskNum, 30
ScaledBxDF, **431**
 bxdf, 431
 f(), 432
 s, 431
ScaleTexture, **520**
 Evaluate(), 521
 tex1, 520
 tex2, 520
Scene, **22**
 aggregate, 23
 bound, 24
 Intersect(), 23
 IntersectP(), 24
 lights, 23
 volumeRegion, 23
 WorldBound(), 24
SDEdge, **155**
 f, 155
 fOedgeNum, 155
 v, 155
SDFace, **153**
 children, 153
 f, 153
 nextFace(), 159
 nextVert(), 159
 otherVert(), 168
 prevFace(), 159
 prevVert(), 159
 v, 153
 vnum(), 159
SDVertex, **152**
 boundary, 152
 child, 152
 oneRing(), 164
 P, 152
 regular, 152
 startFace, 152
 valence(), 158
Shape, **108**
 Area(), 113
 CanIntersect(), 110
 GetShadingGeometry(), 113
 Intersect(), 111
 IntersectP(), 111
 nextshapeId, 109
 ObjectBound(), 109
 ObjectToWorld, 108
 Pdf(), 717
 Refine(), 110
 ReverseOrientation, 108
 Sample(), 716
 shapeId, 109
 TransformSwapsHandedness, 108
 WorldBound(), 110
 WorldToObject, 108
ShapeSet, **626**
 areaDistribution, 723
 areas, 723
 Pdf(), 724
 Sample(), 723
 shapes, 626
 sumArea, 723
SingleScatteringIntegrator, **882**
 Li(), 882
Spectrum, **263**
 Clamp(), 265
 Exp(), 265
 FromRGB(), 277
 HasNaNs(), 266
 IsBlack(), 265
 Lerp(), 265
 Pow(), 265
 Sqrt(), 265
 ToRGB(), 275
 ToXYZ(), 272
 y(), 273
SpectrumType, **277**
SpecularReflection, **440**
 f(), 441
 fresnel, 440
 Pdf(), 703
 R, 440
 Sample_f(), 441
SpecularTransmission, **444**
 etai, 444
 etat, 444
 f(), 444
 fresnel, 444
 Pdf(), 703
 Sample_f(), 444
 T, 444
Sphere, **115**
 Area(), 124
 Bound, 116

Intersect(), 117
IntersectP(), 123
Pdf(), 722
phiMax, 116
radius, 116
Sample(), 720
thetaMax, 116
thetaMin, 116
zmax, 116
zmin, 116
SphericalMapping2D, **516**
Map(), 516
sphere(), 516
WorldToTexture, 516
SpotLight, **612**
cosFalloffStart, 613
cosTotalWidth, 613
Falloff(), 614
Intensity, 613
lightPos, 613
Power(), 614
Sample_L(), 713
StratifiedSampler, **349**
GetMoreSamples(), 351
GetSubSampler(), 350
jitterSamples, 349
MaximumSampleCount(), 359
RoundSize(), 349
sampleBuf, 349
xPixelSamples, 349
xPos, 349
yPixelSamples, 349
yPos, 349
SubsurfaceMaterial, **599**
bumpMap, 599
eta, 599
GetBSDF(), 599
GetBSSRDF(), 600
Kr, 599
scale, 599
sigma_a, 599
sigma_prime_s, 599
SubsurfaceOctreeNode, **898**
children, 899
E, 899
InitHierarchy(), 901
Insert(), 899
ips, 899
isLeaf, 899
Mo(), 911
p, 899
sumArea, 899
SurfaceIntegrator, **740**
Li(), 741
SurfacePoint, **890**
area, 891
n, 891
p, 891
rayEpsilon, 891
SurfacePointsRenderer, **888**
filename, 889
minDist, 889
pCamera, 889
points, 890
Render(), 890
time, 889
Run(), 892
Task, **1041**
Run(), 1041
TexInfo, **525**
Texture, **519**
Evaluate(), 520
TextureMapping2D, **514**
Map(), 514
TextureMapping3D, **519**
Map(), 519
TextureParams, **1061**
FindBool(), 1062
FindFloat(), 1062
FindNormal(), 1062
FindPoint(), 1062
FindSpectrum(), 1062
FindString(), 1062
FindVector(), 1062
floatTextures, 1061
geomParams, 1061
GetSpectrumTexture(), 1062
materialParams, 1061
ReportUnused(), 1062
spectrumTextures, 1061
Transform, **76**
HasScale(), 80
Inverse(), 77
IsIdentity(), 77
m, 76
mInv, 76
operator(), 86
SwapsHandedness(), 89
TransformCache, **1066**
Lookup(), 1066
TransformedPrimitive, **190**
GetAreaLight(), 191
GetBSDF(), 191
GetBSSRDF(), 191
Intersect(), 190
IntersectP(), 191
primitive, 190
WorldBound(), 191
WorldToPrimitive, 190
TransformSet, **1054**
Inverse(), 1054
IsAnimated(), 1064
t, 1054
Triangle, **139**
Area(), 145
GetShadingGeometry(), 145
GetUVs(), 144
Intersect(), 141
mesh, 139
ObjectBound(), 139
Sample(), 719
v, 139
WorldBound(), 140
TriangleFilter, **396**
Evaluate(), 396
TriangleMesh, **135**
alphaTexture, 137
CanIntersect(), 138
n, 137
ntris, 137
nverts, 137
ObjectBound(), 138
p, 137
Refine(), 139
s, 137
uvs, 137
vertexIndex, 137
WorldBound(), 138
UseRadianceProbes, **965**
bbox, 965
c_in, 965
includeDirectInProbes, 965
includeIndirectInProbes, 965
Li(), 965
lmax, 965
nProbes, 965
UVMapping2D, **515**
du, 515
dv, 515
Map(), 515
su, 515
sv, 515
UVTexture, **545**
mapping, 545
Vector, **57**
HasNaNs(), 58
Length(), 62
LengthSquared(), 62
Normalize(), 63
VirtualLight, **779**
n, 779
p, 779
pathContrib, 779
rayEpsilon, 779
VisibilityTester, **608**
r, 608
SetRay(), 609
SetSegment(), 609

VisibilityTester *(continued)*
Transmittance(), 609
Unoccluded(), 609
VolumeGridDensity, **593**
D(), 594
density, 593
Density(), 593
extent, 593
nx, 593
ny, 593
nz, 593
VolumeIntegrator, **876**
Li(), 876
Transmittance(), 876
VolumeRegion, **587**
IntersectP(), 588
Lve(), 588
p(), 588
sigma_a(), 588
sigma_s(), 588
sigma_t(), 588
tau(), 588
WorldBound(), 587
Voxel, **202**
AddPrimitive(), 202
allCanIntersect, 202
Intersect(), 206
primitives, 202
WhittedIntegrator, **42**
Li(), 42
maxDepth, 42
WindyTexture, **567**
Evaluate(), 567
mapping, 567
WrinkledTexture, **566**

Σ INDEX OF MISCELLANEOUS IDENTIFIERS

Finally, this index covers functions, module-local variables, preprocessor definitions, and other miscellaneous identifiers used in the system.

AbsCosTheta(), 426
AbsDot(), 61
activeTransformBits, 1054
ALL_TRANSFORMS_BITS, 1054
alloca(), 1009
AllocAligned(), 1013
Assert(), 1005
AtomicAdd(), 1038
AtomicCompareAndSwap(), 1037
AverageSpectrumSamples(), 268
BalanceHeuristic(), 693
BRDFRemap(), 465
BSDF_ALL, 428
BSDF_ALL_REFLECTION, 428
BSDF_ALL_TRANSMISSION, 428
BSDF_ALL_TYPES, 428
BSDF_ALLOC(), 482
BSDF_DIFFUSE, 428
BSDF_GLOSSY, 428
BSDF_REFLECTION, 428
BSDF_SPECULAR, 428
BSDF_TRANSMISSION, 428
BSSRDF, 598
Ceil2Int(), 1002
CIE_lambda, 271
CIE_X, 271
CIE_Y, 271
CIE_Z, 271
Clamp(), 1000
ComputeLightSamplingCDF(), 709
ConcentricSampleDisk(), 667
CoordinateSystem(), 63
CosineHemispherePdf(), 670
CosineSampleHemisphere(), 669
CosPhi(), 427
CosTheta(), 426
CreateSphereShape(), 1073
Cross(), 62
currentApiState, 1052
curTransform, 1054
Degrees(), 1001
Distance(), 65
DistanceSquared(), 65
divfact(), 940
Dot(), 60
END_TRANSFORM_BITS, 1054
EnqueueTasks(), 1041
EPhoton(), 832
Error(), 1005
EstimateDirect(), 749
Faceforward(), 66
FBm(), 561
Fdr(), 905
FILTER_TABLE_SIZE, 407
FindPoissonPointDistribution(), 896
Float2Int(), 1002
Floor2Int(), 1002
FOR_ACTIVE_TRANSFORMS(), 1055
FrCond(), 435
FrDiel(), 435
FreeAligned(), 1013
Gaussian(), 398
GeneratePath(), 843
GeneratePermutation(), 366
GetVolumeScatteringProperties(), 600
Grad(), 557
graphicsState, 1060
HGPdf(), 731
I(), 856
ImageWrap, 530
INFINITY, 1002
Info(), 1005
InterpolateSpectrumSamples(), 281
INV_FOURPI, 1002
INV_PI, 1002
INV_TWOPI, 1002
Inverse(), 1021
IsPowerOf2(), 1001
K(), 939
kernel(), 830
lambda(), 968
Lanczos(), 533
LargeStep(), 840
LatinHypercube(), 356
LDPixelSample(), 375
LDPixelSampleFloatsNeeded(), 374
LDShuffleScrambled1D(), 377
LDShuffleScrambled2D(), 377
legendrep(), 936

Lerp(), 1000
Log2(), 1001
Log2Int(), 1001
LookAt(), 84
LPhoton(), 831
M_PI, 1002
main(), 20
MakeLight(), 1073
MakeShape(), 1066
MAX_TRANSFORMS, 1054
Mod(), 1001
mutate(), 841
namedCoordinateSystems, 1055
nCIESamples, 271
NEXT(), 153
Noise(), 555
NOISE_PERM_SIZE, 557
NoisePerm, 557
NoiseWeight(), 558
nRGB2SpectSamples, 276
nSpectralSamples, 266
NumSystemCores(), 1041
octreeChildBound(), 1026
Orthographic(), 307
ParseFile(), 21
PBRT_L1_CACHE_LINE_SIZE, 1012
PBRT_VERSION, 1000
pbrtAccelerator(), 1058
pbrtActiveTransformAll(), 1057
pbrtActiveTransformEndTime(), 1057
pbrtActiveTransformStartTime(), 1057
pbrtAreaLightSource(), 1065
pbrtAttributeBegin(), 1060
pbrtAttributeEnd(), 1060
pbrtCamera(), 1058
pbrtCleanup(), 1052
pbrtConcatTransform(), 1055
pbrtCoordinateSystem(), 1055
pbrtCoordSysTransform(), 1056
pbrtFilm(), 1058
pbrtIdentity(), 1054
pbrtInit(), 1051
pbrtLightSource(), 1065
pbrtLookAt(), 1055
pbrtMakeNamedMaterial(), 1064
pbrtMaterial(), 1064
pbrtNamedMaterial(), 1064
pbrtObjectBegin(), 1069
pbrtObjectEnd(), 1069
pbrtObjectInstance(), 1070
pbrtPixelFilter(), 1057
pbrtRenderer(), 1058
pbrtRotate(), 1055
pbrtSampler(), 1058
pbrtScale(), 1055
pbrtShape(), 1065
pbrtSurfaceIntegrator(), 1058
pbrtTexture(), 1063
pbrtTransform(), 1055
pbrtTransformBegin(), 1061
pbrtTransformEnd(), 1061
pbrtTransformTimes(), 1057
pbrtTranslate(), 1055
pbrtVolumeIntegrator(), 1058
pbrtWorldBegin(), 1059
pbrtWorldEnd(), 1071
PermutedRadicalInverse(), 366
Perspective(), 311
PhaseHG(), 585
PhaseIsotropic(), 584
PhaseMieHazy(), 585
PhaseMieMurky(), 585
PhaseRayleigh(), 585
PhaseSchlick(), 586
PowerHeuristic(), 693
PREV(), 153
primes, 368
primsO, 240
prims1, 240
pushedActiveTransformBits, 1060
pushedGraphicsStates, 1060
pushedTransforms, 1060
Quadratic(), 118
Radians(), 1001
RadicalInverse(), 362
RdIntegral(), 912
RdToAlphap(), 912
ReadFloatFile(), 1006
ReadImage(), 1004
RejectionSampleDisk(), 652
renderOptions, 1056
RGBToXYZ(), 274
RNG, 1003
Rotate(), 83
RotateX(), 81
RotateY(), 82
RotateZ(), 82
Round2Int(), 1002
RoundUpPow2(), 1002
SameHemisphere(), 695
SampleO2(), 372
SAMPLE_TABLE_SIZE, 380
sampledLambdaEnd, 266
sampledLambdaStart, 266
SampleHG(), 731
Scale(), 80
Severe(), 1005
SHComputeBSDFMatrix(), 978
SHComputeDiffuseTransfer(), 973
SHComputeTransferMatrix(), 976
SHConvolveCosTheta(), 968
SHConvolvePhong(), 969
SHEvaluate(), 936
SHIndex(), 935
SHMatrixVectorMultiply(), 981
SHProjectCube(), 946
SHProjectIncidentDirectRadiance(), 949
SHProjectIncidentIndirect Radiance(), 950
SHReduceRinging(), 951
SHRotate(), 953
SHRotateXMinus(), 956
SHRotateXPlus(), 956
SHRotateZ(), 955
SHTerms(), 935
Shuffle(), 354
sinCosIndexed(), 940
SinPhi(), 427
SinTheta(), 426
SinTheta2(), 426
Slerp(), 96
SmallStep(), 842
SmoothStep(), 563
Sobol2(), 372
SolveLinearSystem2x2(), 1020
SortSpectrumSamples(), 267
SPECTRUM_ILLUMINANT, 277
SPECTRUM_REFLECTANCE, 277
SpectrumSamplesSorted(), 267
SpecularReflect(), 46
SpecularTransmit(), 47
SphericalDirection(), 292
SphericalPhi(), 292
SphericalTheta(), 292
SQRT_SAMPLE_TABLE_SIZE, 380
START_TRANSFORM_BITS, 1054
STATE_OPTIONS_BLOCK, 1052
STATE_UNINITIALIZED, 1052
STATE_WORLD_BLOCK, 1052
StratifiedSample1D(), 352
StratifiedSample2D(), 352
SubsurfaceFromDiffuse(), 913
TasksCleanup(), 1041
TasksInit(), 1041
TEXTURE_BLACK, 530
TEXTURE_CLAMP, 530
TEXTURE_REPEAT, 530
toZYZ(), 953
transformCache, 1066
Translate(), 79
Transpose(), 1021
Turbulence(), 564
UniformConePdf(), 713
UniformHemispherePdf(), 664
UniformSampleAllLights(), 745
UniformSampleCone(), 713

UniformSampleDisk(), 666
UniformSampleHemisphere(), 664
UniformSampleOneLight(), 746
UniformSampleSphere(), 664
UniformSampleTriangle(), 671
UniformSpherePdf(), 664
Union(), 72
VanDerCorput(), 372
VERIFY_INITIALIZED, 1053
VERIFY_OPTIONS(), 1053
VERIFY_WORLD(), 1053
WaitForAllTasks(), 1041
WARN_IF_ANIMATED_TRANSFORM(), 1063
Warning(), 1005
WriteImage(), 1004
XYZToRGB(), 273

This page intentionally left blank

References

Agarwal, S., R. Ramamoorthi, S. Belongie, and H. W. Jensen. 2003. Structured importance sampling of environment maps. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003) 22*(3), 605–12.

Aggarwal, A. 2002. Software caching vs. prefetching. In *Proceedings of the Third International Symposium on Memory Management,* pp. 157–62. ACM Press.

Aila, T., and S. Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High Performance Graphics 2009,* pp. 145–50.

Airey, J., J. Rohlf, and F. Brooks, Jr. 1990. Towards image realism with interactive update rates in complex virtual building environments. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics,* pp. 41–50.

Akenine-Möller, T. 2001. Fast 3D triangle-box overlap testing. *Journal of Graphics Tools 6*(1), 29–33.

Akenine-Möller, T., and J. Hughes. 1999. Efficiently building a matrix to rotate one vector to another. *Journal of Graphics Tools 4*(4). 1–4.

Akenine-Möller, T., E. Haines, and N. Hoffman. 2008. *Real-Time Rendering*. Natick, MA: A. K. Peters.

Amanatides, J. 1984. Ray tracing with cones. *Computer Graphics (SIGGRAPH '84 Proceedings), 18,* 129–35.

Amanatides, J. 1992. Algorithms for the detection and elimination of specular aliasing. In *Graphics Interface '92,* pp. 86–93.

Amanatides, J., and D. P. Mitchell. 1990. Some regularization problems in ray tracing. In *Graphics Interface '90,* pp. 221–28.

Amanatides, J., and A. Woo. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87,* pp. 3–10.

Anderson, S. 2004. *graphics.stanford.edu/~seander/bithacks.html.*

Annen, T., J. Kautz, F. Durand, and H.-P. Seidel. 2004. Spherical harmonic gradients for mid-range illumination. In *Proceedings of Eurographics Symposium on Rendering*. 331–36.

Anton, H. A., I. Bivens, and S. Davis. 2001. *Calculus* (7th ed.). New York: John Wiley & Sons.

Apodaca, A. A., and L. Gritz. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures.* San Francisco: Morgan Kaufmann.

Appel, A. 1968. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference., 32,* 37–45.

Arbree, A., B. Walter, and K. Bala. 2008. Single-pass scalable subsurface rendering with lightcuts. *Computer Graphics Forum (Proceedings of Eurographics 2008) 27*(2), 507–16.

Arikan, O., D. Forsyth, and J. F. O'Brien. 2005. Fast and detailed approximate global illumination by irradiance decomposition. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1108–14.

Arnaldi, B., T. Priol, and K. Bouatouch. 1987. A new space subdivision method for ray tracing CSG modeled scenes. *The Visual Computer 3*(2), 98–108.

Arvo, J. 1986. Backward ray tracing. *Developments in Ray Tracing, SIGGRAPH '86 Course Notes.*

Arvo, J. 1988. Linear-time voxel walking for octrees. *Ray Tracing News 12*(1).

Arvo, J. 1990. Transforming axis-aligned bounding boxes. In A. S. Glassner (Ed.), *Graphics Gems I,* pp. 548–50. San Diego: Academic Press.

Arvo, J. 1993. Transfer equations in global illumination. In *Global Illumination, SIGGRAPH '93 Course Notes,* Volume 42.

Arvo, J. 1995. *Analytic Methods for Simulated Light Transport*. Ph.D. thesis, Yale University.

Arvo, J., and D. Kirk. 1987. Fast ray tracing by ray classification. *Computer Graphics (SIGGRAPH '87 Proceedings), 21,* 55–64.

Arvo, J., and D. Kirk. 1990. Particle transport and image synthesis. *Computer Graphics (SIGGRAPH '90 Proceedings) 24*(4), 63–66.

Ashdown, I. 1993. Near-field photometry: a new approach. *Journal of the Illuminating Engineering Society 22*(1), 163–80.

Ashdown, I. 1994. *Radiosity: A Programmer's Perspective*. New York: John Wiley & Sons.

Ashikhmin, M., and P. Shirley 2000. An Anisotropic Phong Light Reflection Model, Technical report UUCS-00-014. University of Utah.

Ashikhmin, M., and P. Shirley 2002. An anisotropic Phong BRDF model. *Journal of Graphics Tools 5*(2), 25–32.

Ashikhmin, M., S. Premoze, and P. S. Shirley. 2000. A microfacet-based BRDF generator. In *Proceedings of ACM SIGGRAPH 2000,* Computer Graphics Proceedings, Annual Conference Series, pp. 65–74. ACM Press.

Atkinson, K. 1993. *Elementary Numerical Analysis.* New York: John Wiley & Sons.

Badouel, D., and T. Priol. 1989. An efficient parallel ray tracing scheme for highly parallel architectures. In *Fifth Eurographics Workshop on Graphics Hardware*.

Bahar, E., and S. Chakrabarti. 1987. Full-wave theory applied to computer-aided graphics for 3D objects. *IEEE Computer Graphics and Applications 7*(7), 46–60.

Banks, D. C. 1994. Illumination in diverse codimensions. In *Proceedings of SIGGRAPH '94,* Computer Graphics Proceedings, Annual Conference Series, pp. 327–34.

Barkans, A. C. 1997. High-quality rendering using the Talisman architecture. In *1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware,* pp. 79–88.

Barzel, R. 1997. Lighting controls for computer cinematography. *Journal of Graphics Tools 2*(1), 1–20.

Basri, R. and D. Jacobs. 2001. Lambertian reflectance and linear subspaces. In *IEEE International Conference on Computer Vision*. 383–90.

Becker, B. G., and N. L. Max. 1993. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH '93,* Computer Graphics Proceedings, Annual Conference Series, pp. 183–90.

Beckmann, P., and A. Spizzichino. 1963. *The Scattering of Electromagnetic Waves from Rough Surfaces*. New York: Pergamon.

Ben-Artzi, A., K. Egan, F. Durand, and R. Ramamoorthi. 2008. A precomputed polynomial representation for interactive BRDF editing with global illumination. *ACM Transactions on Graphics 27*(2), 13:1–13:13.

Benthin, C., and I. Wald. 2009. Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing. In *Proceedings of High Performance Graphics 2009,* pp. 135-44.

Benthin, C., S. Boulos, D. Lacewell, and I. Wald. 2007. Packet-Based Ray Tracing of Catmull–Clark Subdivision Surfaces, *SCI Institute Technical Report, No. UUSCI-2007-011.* University of Utah.

Benthin, C., I. Wald, and P. Slusallek. 2003. A scalable approach to interactive global illumination. In *Computer Graphics Forum 22*(3). 621–30.

Benthin, C., I. Wald, and P. Slusallek. 2004. Techniques for interactive ray tracing of Bézier surfaces. *Journal of Graphics, GPU, and Game Tools 11*(2), 1–16.

Berger, E. D., B. G. Zorn, and K. S. McKinley. 2001. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation,* pp. 114–24.

Berger, E. D., B. G. Zorn, and K. S. McKinley. 2002. Reconsidering custom memory allocation. In *Proceedings of ACM OOPSLA 2002*.

Betrisey, C., J. F. Blinn, B. Dresevic, B. Hill, G. Hitchcock, B. Keely, D. P. Mitchell, J. C. Platt, and T. Whitted. 2000. Displaced filtering for patterned displays. *Society for Information Display International Symposium. Digest of Technical Papers 31,* 296–99.

Bhate, N., and A. Tokuta. 1992. Photorealistic volume rendering of media with directional scattering. In *Proceedings of the Third Eurographics Rendering Workshop,* pp. 227–45.

Bigler, J., A. Stephens, and S. Parker. 2006. Design for parallel interactive ray tracing systems. *IEEE Symposium on Interactive Ray Tracing,* pp. 187–195.

Bikker, J. and R. Reijerse. 2009. A precalculated point set for caching shading information. *Eurographics Short Papers*.

Bjorke, K. 2001. Using Maya with RenderMan on Final Fantasy: The Spirits Within. *SIGGRAPH 2001 RenderMan Course Notes.*

Blasi, P., B. L. Saëc, and C. Schlick. 1993. A rendering algorithm for discrete volume density objects. *Computer Graphics Forum (Proceedings of Eurographics '93) 12*(3), 201–10.

Blinn, J. F. 1977. Models of light reflection for computer synthesized pictures. *Computer Graphics (SIGGRAPH '77 Proceedings), 11,* 192–98.

Blinn, J. F. 1978. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings), 12,* 286–92.

Blinn, J. F. 1982a. A generalization of algebraic surface drawing. *ACM Transactions on Graphics 1*(3), 235–56.

Blinn, J. F. 1982b. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics 16*(3), 21–29.

Blinn, J. F., and M. E. Newell. 1976. Texture and reflection in computer generated images. *Communications of the ACM 19,* 542–46.

Bloom, C., J. Blow, and C. Muratori. 2004. Errors and omissions in Marc Alexa's "Linear Combination of Transformations". *www.cbloom.com/3d/techdocs/lcot_errors.pdf*

Blow, J. 2004. Understandling slerp, then not using it. *Game Developer Magazine*. Also available from *number-none.com/product/Understanding Slerp, Then Not Using It*

Blumofe, R., and C. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM 46*(5), 720–48.

Blumofe, R., C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. 1996. Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Compututing 37*(1), 55–69.

Boehm, H.-J. 2005. Threads cannot be implemented as a library. *ACM SIGPLAN Notices 40*(6), 261–68.

Bolin, M. R., and G. W. Meyer. 1995. A frequency based ray tracer. In *Proceedings of SIGGRAPH '95,* Computer Graphics Proceedings, Annual Conference Series, pp. 409–18.

Bolin, M. R., and G. W. Meyer. 1998. A perceptually based adaptive sampling algorithm. In *Proceedings of SIGGRAPH '98,* Computer Graphics Proceedings, Annual Conference Series, pp. 299–310.

Bolz, J., and P. Schröder. 2002. Rapid evaluation of Catmull–Clark subdivision surfaces. In *Web3D 2002 Symposium*.

Booth, T. E. 1986. A Monte Carlo learning/biasing experiment with intelligent random numbers. *Nuclear Science and Engineering 92,* 465–81.

Borges, C. 1991. Trichromatic approximation for computer graphics illumination models. In *Computer Graphics (Proceedings of SIGGRAPH '91), 25,* 101–04.

Boulos, S., and E. Haines. 2006. Ray–box sorting. *Ray Tracing News 19*(1), *tog.acm.org/resources/RTNews/html/rtnv19n1.html*.

Boulos, S., I. Wald, and C. Benthin. 2008. Adaptive ray packet reordering. In *Proceedings of IEEE Symposium on Interactive Ray Tracing,* pp. 131–38.

Bracewell, R. N. 2000. *The Fourier Transform and Its Applications*. New York: McGraw-Hill.

Bronsvoort, W. F., and F. Klok. 1985. Ray tracing generalized cylinders. *ACM Transactions on Graphics 4*(4), 291–303.

Buck, R. C. 1978. *Advanced Calculus*. New York: McGraw-Hill.

Budge, B., D. Coming, D. Norpchen, and K. Joy. 2008. Accelerated building and ray tracing of restricted BSP trees. In *IEEE Symposium on Interactive Ray Tracing,* pp. 167–174.

Budge, B., T. Bernardin, J. Stuart, S. Sengupta, K. Joy, and J. D. Owens. 2009. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum (Proceedings of Eurographics 2009) 28*(2), 385–96.

Buhler, J., and D. Wexler. 2002. A phenomenological model for Bokeh rendering. *SIGGRAPH Sketch*. See also *www.flarg.com/bokeh.html.*

Burke, D., A. Ghosh, and W. Heidrich. 2005. Bidirectional importance sampling for direct illumination. In *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering,* 147–56.

Buss, S., J. Fillmore. 2001. Spherical averages and applications to spherical splines and interpolation. *ACM Transactions on Graphics 20*(2), 95–126.

Cabral, B., N. Max, and R. Springmeyer. 1987. Bidirectional reflection functions from surface bump maps. *Computer Graphics (SIGGRAPH '87 Proceedings), 21,* 273–81.

Calder, B., K. Chandra, S. John, and T. Austin. 1998. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII),* San Jose.

Cammarano, M., and H. W. Jensen. 2002. Time dependent photon mapping. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*. 135–44.

Cant, R. J., and P. A. Shrubsole 2000. Texture potential MIP mapping, a new high-quality texture antialiasing algorithm. *ACM Transactions on Graphics 19*(3), 164–84.

Cantrill, B., M. Shapiro, and A. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*.

Carr, N., J. D. Hall, and J. Hart. 2002. The ray engine. In *Proceedings of Graphics Hardware 2002* (September).

Catmull, E., and J. Clark. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10,* 350–55.

Cazals, F., G. Drettakis, and C. Puech. 1995. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. *Computer Graphics Forum 14*(3), 371–82.

Cerezo, E., F. Perez-Cazorla, X. Pueyo, F. Seron, and F. Sillion. 2005. A survey on participating media rendering techniques. *The Visual Computer 21*(5), 303–28.

Chalmers, A., T. Davis, and E. Reinhard. 2002. *Practical Parallel Rendering*. Natick, Massachusetts: A. K. Peters.

Chandrasekhar, S. 1960. *Radiative Transfer*. New York: Dover Publications. Originally published by Oxford University Press, 1950.

Chib, S. and E. Greenberg. 1995. Understanding the Metropolis–Hastings algorithm. *The American Stastician 49*(4). 327–35.

Chilimbi, T. M., B. Davidson, and J. R. Larus. 1999a. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation,* pp. 13–24.

Chilimbi, T. M., M. D. Hill, and J. R. Larus. 1999b. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation,* pp. 1–12.

Chiu, K., P. Shirley, and C. Wang. 1994. Multi-jittered sampling. In P. Heckbert (Ed.), *Graphics Gems IV,* pp. 370–74. San Diego: Academic Press.

Chen, B., F. Dachille, and A. Kaufman. 2004. Footprint area sampled texturing. *IEEE Transactions on Visualization and Computer Graphics 10*(2), 230–40.

Christensen, P. H. 1999. Faster photon map global illumination. *Journal of Graphics Tools 4*(3), 1–10.

Christensen, P. H. 2003. Adjoints and importance in rendering: an overview. *IEEE Transactions on Visualization and Computer Graphics 9*(3), 329–40.

Christensen, P., and D. Batali. 2004. An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*. 133–42.

Christensen, P. H., D. M. Laur, J. Fong, W. L. Wooten, and D. Batali. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Computer Graphics Forum (Eurographics 2003 Conference Proceedings),* pp. 543–52.

Chvolson, O. D. 1890. Grundzüge einer matematischen Theorie der inneren Diffusion des Lichtes. *Izv. Peterburg. Academii Nauk 33,* 221–65.

Clarberg, P. 2008. Fast equal-area mapping of the (hemi)sphere using SIMD. *Journal of Graphics Tools 13*(3), 53–68.

Clarberg, P., and T. Akenine-Möller. 2008a. Practical product importance sampling for direct illumination. *Computer Graphics Forum (Proceedings of Eurographics 2008) 27*(2), 681–90.

Clarberg, P., and T. Akenine-Möller. 2008b. Exploiting visibility correlation in direct illumination. *Computer Graphics Forum 27*(4), 1125–36.

Clarberg, P., W. Jarosz, T. Akenine-Möller, and H. W. Jensen. 2005. Wavelet importance sampling: efficiently evaluating products of complex functions. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1166–75.

Clark, J. H. 1976. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*(10), 547–54.

Cleary, J. G., and G. Wyvill. 1988. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer 4*(2), 65–83.

Cleary, J. G., B. M. Wyvill, R. Vatti, and G. M. Birtwistle. 1983. Design and analysis of a parallel ray tracing computer. In *Graphics Interface '83,* pp. 33–38.

Cline, D., D. Adams, and P. Egbert. 2008. Table-driven adaptive importance sampling. *Computer Graphics Forum (Proceedings of the 2008 Eurographics Symposium on Rendering) 27*(4), 1115–23.

Cline, D., J. Talbot, and P. Egbert. 2005. Energy redistribution path tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1186–95.

Cline, D., P. Egbert, J. Talbot, and D. Cardon. 2006. Two stage importance sampling for direct lighting. *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering,* 103–14.

Cline, D., S. Jeschke, K. White, A. Razdan, and P. Wonka. 2009a. Dart throwing on surfaces. In *Computer Graphics Forum (Proceedings of the 20th Eurographics Symposium on Rendering) 28*(4), 1217–26.

Cline, D., A. Razdan, and P. Wonka. 2009b. A comparison of tabular PDF inversion methods. *Computer Graphics Forum 28*(1), 154–60.

Cohen, J., A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks Jr., and W. Wright. 1996. Simplification envelopes. In *Proceedings of SIGGRAPH '96,* Computer Graphics Proceedings, Annual Conference Series, pp. 119–28.

Cohen, M., and J. Wallace. 1993. *Radiosity and Realistic Image Synthesis*. San Diego: Academic Press Professional.

Collins, S. 1994. Adaptive splatting for specular to diffuse light transport. In *Fifth Eurographics Workshop on Rendering,* Darmstadt, Germany, pp. 119–35.

Contini, D., F. Martelli, and G. Zaccanti. 1997. Photon migration through a turbid slab described by a model based on diffusion approximation. I. Theory. *Applied Optics 36*(19), 4587–4599.

Cook, R. L. 1984. Shade trees. *Computer Graphics (SIGGRAPH '84 Proceedings), 18,* 223–31.

Cook, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*(1), 51–72.

Cook, R., and T. DeRose. 2005. Wavelet noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 803–11.

Cook, R. L., and K. E. Torrance. 1981. A reflectance model for computer graphics. *Computer Graphics (SIGGRAPH '81 Proceedings), 15,* 307–16.

Cook, R. L., and K. E. Torrance. 1982. A reflectance model for computer graphics. *ACM Transactions on Graphics 1*(1), 7–24.

Cook, R. L., T. Porter, and L. Carpenter. 1984. Distributed ray tracing. *Computer Graphics (SIGGRAPH '84 Proceedings), 18,* 137–45.

Cook, R. L., L. Carpenter, and E. Catmull. 1987. The REYES image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH '87),* 95–102.

Crow, F. C. 1977. The aliasing problem in computer-generated shaded images. *Communications of the ACM 20*(11), 799–805.

Crow, F. C. 1984. Summed-area tables for texture mapping. *Computer Graphics (Proceedings of SIGGRAPH '84), 18,* 207–12.

Dachsbacher, C., and M. Stamminger. 2003. Translucent shadow maps. In *Proceedings of the 13th Eurographics Workshop on Rendering,* pp. 197–201.

Dachsbacher, C., and M. Stamminger. 2006. Splatting of indirect illumination. *ACM Symposium on Interactive 3D Graphics and Games*m 93–100.

Dammertz, H., and A. Keller. 2006. Improving ray tracing precision by object space intersection computation. *IEEE Symposium on Interactive Ray Tracing*. pp. 25–31.

Dammertz, H., and A. Keller. 2008. The edge volume heuristic—robust triangle subdivision for improved BVH performance, In *IEEE Symposium on Interactive Ray Tracing*. pp. 155–58.

Dana, K. J., B. van Ginneken, S. K. Nayar, and J. J. Koenderink. 1999. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics 18*(1), 1–34.

Danskin, J., and P. Hanrahan. 1992. Fast algorithms for volume ray tracing. In *1992 Workshop on Volume Visualization,* pp. 91–98.

de Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf. 2000. *Computational Geometry: Algorithms and Applications*. New York: Springer-Verlag.

de Voogt, E., A. van der Helm, and W. F. Bronsvoort. 2000. Ray tracing deformed generalized cylinders. *The Visual Computer 16*(3–4), 197–207.

Debevec, P. 1998. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of SIGGRAPH '98,* 189–98.

Deering, M. F. 1995. Geometry compression. In *Proceedings of SIGGRAPH '95,* Computer Graphics Proceedings, Annual Conference Series, pp. 13–20.

d'Eon, E., D. Luebke, and E. Enderton. 2007. Efficient rendering of human skin. In *Rendering Techniques 2007: 18th Eurographics Workshop on Rendering,* 147–58.

DeRose, T. D. 1989. *A Coordinate-Free Approach to Geometric Programming*. *Math for SIGGRAPH, SIGGRAPH Course Notes #23.* Also available as Technical Report No. 89-09-16, Department of Computer Science and Engineering, University of Washington, Seattle.

Deussen, O., P. M. Hanrahan, B. Lintermann, R. Mech, M. Pharr, and P. Prusinkiewicz. 1998. Realistic modeling and rendering of plant ecosystems. In *Proceedings of SIGGRAPH '98,* Computer Graphics Proceedings, Annual Conference Series, pp. 275–86.

Devlin, K., A. Chalmers, A. Wilkie, and W. Purgathofer. 2002. Tone reproduction and physically based spectral rendering. In D. Fellner and R. Scopignio (Eds.), *Proceedings of Eurographics 2002,* pp. 101–23. The Eurographics Association.

Dippé, M. A. Z., and E. H. Wold. 1985. Antialiasing through stochastic sampling. *Computer Graphics (SIGGRAPH '85 Proceedings), 19,* 69–78.

Dobashi, Y., K. Kaneda, H. Nakatani, and H. Yamashita. 1995. A quick rendering method using basis functions for interactive lighting design. *Computer Graphics Forum 14*(3), 229–40.

Dobkin, D. P., and D. P. Mitchell. 1993. Random-edge discrepancy of supersampling patterns. In *Graphics Interface '93,* Toronto, Ontario, pp. 62–69. Canadian Information Processing Society.

Dobkin, D. P., D. Eppstein, and D. P. Mitchell. 1996. Computing the discrepancy with applications to supersampling patterns. *ACM Transactions on Graphics 15*(4), 354–76.

Donikian, M., B. Walter, K. Bala, S. Fernandez, and D. P. Greenberg. 2006. Accurate direct illumination using iterative adaptive sampling. *IEEE Transactions on Visualization and Computer Graphics 12*(3), 353–64.

Donner, C. 2006. *Towards Realistic Image Synthesis of Scattering Materials*. Ph.D. thesis, University of California, San Diego.

Donner, C., and H. W. Jensen. 2005. Light diffusion in multi-layered translucent materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1032–39.

Donner, C., and H. W. Jensen. 2006. A Spectral BSSRDF for shading human skin. *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering,* 409–417.

Donner, C., T. Weyrich, E. d'Eon, R. Ramamoorthi, and S. Rusinkiewicz. 2008. A layered, heterogeneous reflectance model for acquiring and rendering human skin. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2008) 27*(5), 140:1–140:12.

Donner, C., J. Lawrence, R. Ramamoorthi, T. Hachisuka, H. W. Jensen, and S. Nayar. 2009. An empirical BSSRDF model. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009) 28*(3), 30:1–30:10.

Doo, D., and M. Sabin. 1978. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design 10,* 356–60.

Dorsey, J. O., F. X. Sillion, and D. P. Greenberg. 1991. Design and simulation of opera lighting and projection effects. In *Computer Graphics (Proceedings of SIGGRAPH '91), 25,* 41–50.

Dorsey, J., J. Arvo, and D. Greenberg. 1995. Interactive design of complex time-dependent lighting. *IEEE Computer Graphics and Applications 15*(2), 26–36.

Dorsey, J., H. K. Pedersen, and P. M. Hanrahan. 1996. Flow and changes in appearance. In *Proceedings of SIGGRAPH '96,* Computer Graphics Proceedings, Annual Conference Series, pp. 411–20.

Dorsey, J., A. Edelman, J. Legakis, H. W. Jensen, and H. K. Pedersen. 1999. Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH '99,* Computer Graphics Proceedings, Annual Conference Series, pp. 225–34.

Dorsey, J., H. Rushmeier, and F. Sillion. 2008. *Digital Modeling of Material Appearance.* San Francisco: Morgan Kaufmann.

Drebin, R. A., L. Carpenter, and P. Hanrahan. 1988. Volume rendering. *Computer Graphics (Proceedings of SIGGRAPH '88), 22,* 65–74.

Drepper, U. 2007. What Every Programmer Should Know About Memory. *people.redhat.com/drepper/cpumemory.pdf.*

Drew, M. and G. Finlayson. 2003. Multispectral rendering without spectra. *Journal of the Optical Society of America A 20*(7), 1181–1193.

Driemeyer, T., and R. Herken. 2002. *Programming mental ray*. Wien: Springer-Verlag.

Duff, T. 1985. Compositing 3-D rendered images. *Computer Graphics (Proceedings of SIGGRAPH '85), 19,* 41–44.

Dunbar, D., and G. Humphreys. 2006. A spatial data structure for fast Poisson-disk sample generation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) 25*(3), 503–08.

Dungan, W. Jr., A. Stenger, and G. Sutty. 1978. Texture tile considerations for raster graphics. *Computer Graphics (Proceedings of SIGGRAPH '78), 12,* 130–34.

Dutré, P. 2003. Global Illumination Compendium. *www.cs.kuleuven.ac.be/~phil/GI/.*

Dutré, P., K. Bala, and P. Bekaert. 2006. *Advanced Global Illumination*. Natick, Massachusetts: A. K. Peters.

Eberly, D. H. 2001. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. San Francisco: Morgan Kaufmann.

Ebert, D., F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. 2003. *Texturing and Modeling: A Procedural Approach*.

Edwards, D., S. Boulos, J. Johnson, P. Shirley, M. Ashikhmin, M. Stark, and C. Wyman. 2005. The halfway vector disk for brdf modeling. *ACM Transactions on Graphics 25*(1), 1–18.

Egan, K., Y.-T. Tseng, N. Holzschuch, F. Durand, and R. Ramamoorthi. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009) 28*(3), 93:1–93:13.

Eisemann, M., M. Magnor, T. Grosch, and S. Müller. 2007. Fast ray/axis-aligned bounding box overlap tests using ray slopes. *Journal of Graphics, GPU, and Game Tools 12*(4), 35–46.

Ericson, C. 2004. *Real-Time Collision Detection*. Morgan Kaufmann Series in Interactive 3D Technology. San Francisco: Morgan Kaufmann.

Ernst, M. and G. Greiner. 2007. Early split clipping for bounding volume hierarchies. *IEEE Symposium on Interactive Ray Tracing*. pp. 73–78.

Evans, G. and M. McCool. 1999. Stratified wavelength clusters for efficient spectral Monte Carlo rendering. In *Graphics Interface '99,* pp. 42–49.

Fabianowski, B. and J. Dingliana. 2009. Interactive global photon mapping. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering) 28*(4), 1151–59.

Fabianowski, B., C. Fowler, and J. Dingliana. 2009. A cost metric for scene-interior ray origins. In *Short Paper Proceedings of the 30th Annual Conference of the European Association for Computer Graphics (Eurographics 2009),* pp. 49–50.

Fairchild, M. D., and D. R. Wyble. 1998. Colorimetric Characterization of the Apple Studio Display (Flat Panel LCD). Technical report, RIT Munsell Color Science Laboratory.

Fan, S., S. Chenney, and Y.-C. Lai. 2005. Metropolis photon sampling with optional user guidance. In *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering,* 127–38.

Fante, R. L. 1981. Relationship between radiative-transport theory and Maxwell's equations in dielectric media. *Journal of the Optical Society of America 71*(4), 460-468.

Farrell, T., M. Patterson, and B. Wilson. 1992. A diffusion theory model of spatially resolved, steady-state diffuse reflectance for the noninvasive determination of tissue optical properties *in vivo*. *Med. Phys. 19*(4), 879–888.

Fatahalian, K. 2008. Running code at a teraflop: how GPU shader cores work. In *Beyond Programmable Shading, SIGGRAPH 2008 Course Notes. s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf*

Fattal, R. 2009. Participating media illumination using light propagation maps. *ACM Transactions on Graphics 28*(1)., 7:1–7:11.

Fedkiw, R., J. Stam, and H. W. Jensen. 2001. Visual simulation of smoke. In *Proceedings of ACM SIGGRAPH 2001,* Computer Graphics Proceedings, Annual Conference Series, pp. 15–22.

Feibush, E. A., M. Levoy, and R. L. Cook. 1980. Synthetic texturing using digital filters. *Computer Graphics (Proceedings of SIGGRAPH '80), 14,* 294–301.

Fernandez, S., K. Bala, and D. P. Greenberg. 2002. Local illumination environments for direct lighting acceleration. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering,* pp. 7–14.

Ferwerda, J. A. 2001. Elements of early vision for computer graphics. *IEEE Computer Graphics and Applications 21*(5), 22–33.

Fisher, M., K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan. 2009. DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009) 28*(5).

Fishman, G. S. 1996. *Monte Carlo: Concepts, Algorithms, and Applications*. New York: Springer-Verlag.

Fleischer, K., D. Laidlaw, B. Currin, and A. H. Barr. 1995. Cellular texture generation. In *Proceedings of SIGGRAPH '95,* Computer Graphics Proceedings, Annual Conference Series, pp. 239–48.

Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes. 1990. *Computer Graphics: Principles and Practice*. Reading, Massachusetts: Addison-Wesley.

Fournier, A. 1992. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination,* pp. 45–52.

Fournier, A., and E. Fiume. 1988. Constant-time filtering with space-variant kernels. *Computer Graphics (SIGGRAPH '88 Proceedings), 22,* 229–38.

Fournier, A., D. Fussel, and L. Carpenter. 1982. Computer rendering of stochastic models. *Communications of the ACM 25*(6), 371–84.

Fraser, C., and D. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation*. Reading, Massachusetts: Addison-Wesley.

Friedel, I., and A. Keller. 2000. Fast generation of randomized low discrepancy point sets. In *Monte Carlo and Quasi-Monte Carlo Methods 2000,* pp. 257–73. Berlin: Springer-Verlag.

Frisvad, J., N. Christensen, and H. W. Jensen. 2007. Computing the scattering properties of participating media using Lorenz-Mie theory. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26*(3), 60:1–60:10.

Fujimoto, A., T. Tanaka, and K. Iwata. 1986. Arts: accelerated ray-tracing system. *IEEE Computer Graphics and Applications 6*(4), 16–26.

Gardner, G. Y. 1984. Simulation of natural scenes using textured quadric surfaces. *Computer Graphics (SIGGRAPH '84 Proceedings), 18,* 11–20.

Gardner, G. Y. 1985. Visual simulation of clouds. *Computer Graphics (Proceedings of SIGGRAPH '85), 19,* 297–303.

Gardner, R. P., H. K. Choi, M. Mickael, A. M. Yacout, Y. Yin, and K. Verghese. 1987. Algorithms for forcing scattered radiation to spherical, planar circular, and right circular cylindrical detectors for Monte Carlo simulation. *Nuclear Science and Engineering 95,* 245–56.

Gassenbauer, V., J. Křivánek, and K. Bouatouch. 2009. Spatial directional radiance caching. *Computer Graphics Forum (Proceedings of The Eurographics Symposium on Rendering) 28*(4).

Gautron, P., K. Bouatouch, and S. Pattanaik. 2007. Temporal Radiance Caching. *IEEE Transactions on Visualization and Computer Graphics 13*(5). 891–901.

Gershbein, R., and P. M. Hanrahan. 2000. A fast relighting engine for interactive cinematic lighting design. In *Proceedings of ACM SIGGRAPH 2000,* Computer Graphics Proceedings, Annual Conference Series, pp. 353–58.

Gershun, A. 1939. The light field. *Journal of Mathematics and Physics 18,* 51–151.

Génevaux, O., F. Larue, J.-M. Dischler. 2006. Interactive refraction on complex static geometry using spherical harmonics. In *Proceedings of the S006 symposium on Interactive 3D graphics and games,* 145–52.

Georgiev, I., and P. Slusallek. 2008. RTfact: generic concepts for flexible and high performance ray tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing,* 115–22.

Ghosh, A., and W. Heidrich. 2006. Correlated visibility sampling for direct illumination. *The Visual Computer 22*(9–10). 693–701.

Ghosh, A., A. Doucet, and W. Heidrich. 2006. Sequential sampling for dynamic environment map illumination. In *Proceedings of the Eurographics Symposium on Rendering,* 115–26.

Ghosh, A., T. Hawkins, P. Peers, S. Frederiksen, and P. Debevec. 2008. Practical modeling and acquisition of layered facial reflectance. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2008) 27*(5), 139:1–139:10.

Gibson, J. E., and M. D. Fairchild. 2000. Colorimetric characterization of three computer displays (LCD and CRT). Technical report, RIT Munsell Color Science Laboratory, *www.cis.rit.edu/research/mcsl/research/reports.shtml.*

Glassner, A. 1984. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*(10), 15–22.

Glassner, A. (Ed.) 1989a. *An Introduction to Ray Tracing*. San Diego: Academic Press.

Glassner, A. 1989b. How to derive a spectrum from an RGB triplet. *IEEE Computer Graphics and Applications 9*(4), 95–99.

Glassner, A. 1993. Spectrum: an architecture for image synthesis, research, education, and practice. *Developing Large-Scale Graphics Software Toolkits, SIGGRAPH '93 Course Notes, 3,* 1-14–1-43.

Glassner, A. 1994. A model for fluorescence and phosphorescence. In *Proceedings of the Fifth Eurographics Workshop on Rendering*. pp. 57–68.

Glassner, A. 1995. *Principles of Digital Image Synthesis*. San Francisco: Morgan Kaufmann.

Glassner, A. 1999. An open and shut case. *IEEE Computer Graphics and Applications 19*(3), 82–92.

Goesele, M. H. Lensch, J. Lang, C. Fuchs, and H.-P. Seidel. 2004. DISCO–Acquisition of translucent objects. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004) 23*(3), 844–53.

Goldberg, A., M. Zwicker, and F. Durand. 2008. Anisotropic noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27*(3), 54:1–54:8.

Goldman, D. B. 1997. Fake fur rendering. In *Proceedings of SIGGRAPH '97,* Computer Graphics Proceedings, Annual Conference Series, pp. 127–34.

Goldman, R. 1985. Illicit expressions in vector algebra. *ACM Transactions on Graphics 4*(3), 223–43.

Goldsmith, J., and J. Salmon. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*(5), 14–20.

Goldstein, R. A., and R. Nagel. 1971. 3-D visual simulation. *Simulation 16*(1), 25–31.

Gortler, S. J., R. Grzeszczuk, R. Szeliski, and M. F. Cohen. 1996. The lumigraph. In *Proceedings of SIGGRAPH '96,* Computer Graphics Proceedings, Annual Conference Series, pp. 43–54.

Gray, A. 1993. *Modern Differential Geometry of Curves and Surfaces*. Boca Raton, Florida: CRC Press.

Green, S. A., and D. J. Paddon. 1989. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications 9*(6), 12–26.

Greenberg, D. P., K. E. Torrance, P. S. Shirley, J. R. Arvo, J. A. Ferwerda, S. Pattanaik, E. P. F. Lafortune, B. Walter, S.-C. Foo, and B. Trumbore. 1997. A framework for realistic image synthesis. In *Proceedings of SIGGRAPH '97,* Computer Graphics Proceedings, Annual Conference Series, pp. 477–94.

Greene, N. 1986. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications 6*(11), 21–29.

Greene, N., and P. S. Heckbert. 1986. Creating raster Omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications 6*(6), 21–27.

Greger, G., P. Shirley, P. Hubbard, and D. Greenberg. 1998. The irradiance volume. *IEEE Computer Graphics and Applications 18*(2), 32–43.

Gribble, C., and K. Ramani. 2008. Coherent ray tracing via stream filtering. In *Proceedings of IEEE Symposium on Interactive Ray Tracing,* pp. 59–66.

Gritz, L., and E. d'Eon. 2008. The importance of being linear. In H. Nguyen (Ed.), *GPU Gems 3*. Boston, Massachusetts: Addison-Wesley.

Gritz, L., and J. K. Hahn. 1996. BMRT: a global illumination implementation of the RenderMan standard. *Journal of Graphics Tools 1*(3), 29–47.

Grunwald, D., B. G. Zorn, and R. Henderson. 1993. Improving the cache locality of memory allocation. In *SIGPLAN Conference on Programming Language Design and Implementation,* pp. 177–86.

Guatron, P., J. Křivánek, S.-N. Pattanaik, and K. Boutaouch. 2004. A novel hemispherical basis for accurate and efficient rendering. In *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*. 321–330.

Günther, J., T. Chen, M. Goesele, I. Wald, and H.-P. Seidel. 2005. Efficient acquisition and realistic rendering of car paint. In *Proceedings of Vision, Modeling, and Visualization (VMV),* 487–94.

Günther, J., S. Popov, H. P. Seidel, and P. Slusallek. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In *IEEE Symposium on Interactive Ray Tracing,* pp. 113–18.

Guthe, S., and Heckbert, P. 2005. *Non-Power-of-Two Mipmapping, NVIDIA Technical Report, developer.nvidia.com/object/np2_mipmapping.html.*

Haber, T., T. Mertens, P. Bekaert, and F. Van Reeth. 2005. A computational approach to simulate subsurface light diffusion in arbitrarily shaped objects. In *Proceedings of Graphics Interface,* 79–86.

Hachisuka, T. 2005. High-quality global illumination rendering using rasterization. In M. Pharr (Ed.), *GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation*.

Hachisuka, T., W. Jarosz, R. P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker, and H. W. Jensen. 2008a. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27*(3), 33:1–33:10.

Hachisuka, T., S. Ogaki, and H. W. Jensen. 2008b. Progressive Photon Mapping. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2008) 27*(5), 130:1–130:8.

Haines, E. A. 1987. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications 7*(11), 3–5.

Haines, E. A. 1994. Point in polygon strategies. In P. Heckbert (Ed.), *Graphics Gems IV,* pp. 24–46. San Diego: Academic Press.

Haines, E. A., and D. P. Greenberg. 1986. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics and Applications 6*(9), 6–16.

Haines, E. A., and J. R. Wallace. 1994. Shaft culling for efficient ray-traced radiosity. In *Second Eurographics Workshop on Rendering (Photorealistic Rendering in Computer Graphics).* Also in *SIGGRAPH 1991 Frontiers in Rendering Course Notes.*

Hakura, Z. S., and A. Gupta. 1997. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th International Symposium on Computer Architecture,* Denver, Colorado, pp. 108–20.

Hall, R. 1989. *Illumination and Color in Computer Generated Imagery*. New York: Springer-Verlag.

Hall, R. 1999. Comparing spectral color computation methods. *IEEE Computer Graphics and Applications 19*(4), 36–46.

Hall, R. A., and D. P. Greenberg. 1983. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications 3,* 10–20.

Hammersley, J., and D. Handscomb. 1964. *Monte Carlo Methods*. New York: John Wiley.

Han, C., B. Sun, R. Ramamoorthi, and E. Grinspun. 2007. Frequency domain normal map filtering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26*(3), 28:1–28:11.

Hanrahan, P. 1983. Ray tracing algebraic surfaces. *Computer Graphics (Proceedings of SIGGRAPH '83), 17,* 83–90.

Hanrahan, P., and W. Krueger. 1993. Reflection from layered surfaces due to subsurface scattering. In *Computer Graphics (SIGGRAPH '93 Proceedings),* pp. 165–74.

Hanrahan, P., and J. Lawson. 1990. A language for shading and lighting calculations. *Computer Graphics (SIGGRAPH '90 Proceedings), 24,* 289–98.

Hansen, J. E., and L. D. Travis. 1974. Light scattering in planetary atmospheres. *Space Science Reviews 16,* 527–610.

Hao, X., T. Baby, and A. Varshney. 2003. Interactive subsurface scattering for translucent meshes. In *ACM Symposium on Interactive 3D Graphics,* pp. 75–82.

Hart, D., P. Dutré, and D. P. Greenberg. 1999. Direct illumination with lazy visibility evaluation. In *Proceedings of SIGGRAPH '99,* Computer Graphics Proceedings, Annual Conference Series, pp. 147–54.

Hart, J. C. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*(9), 527–45.

Hart, J. C., D. J. Sandin, and L. H. Kauffman. 1989. Ray tracing deterministic 3-D fractals. *Computer Graphics (Proceedings of SIGGRAPH '89), 23,* 289–96.

Hasenfratz, J.-M., M. Lapierre, N. Holzschuch, F. Sillion. 2003. A survey of real-time soft shadows algorithms. *Computer Graphics Forum (Eurographics State of the Art Report) 22*(4), 753–74.

Havran, V., and J. Bittner. 2002. On improving kd-trees for ray shooting. In *Proceedings of WSCG 2002 Conference,* pp. 209–17.

Havran, V., J. Bittner, R. Herzog, and H.-P. Seidel. 2005a. Ray maps for global illumination. In *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering,* pp. 43–54.

Havran, V., R. Herzog, and H.-P. Seidel. 2005b. Fast final gathering via reverse photon mapping, *Computer Graphics Forum (Proceedings of Eurographics 2005) 24*(3), 323–34.

Havran, V., R. Herzog, and H.-P. Seidel. 2006. On the fast construction of spatial hierarchies for ray tracing. In *IEEE Symposium on Interactive Ray Tracing,* pp. 71–80.

Hawkins, T., P. Einarsson, and P. Debevec. 2005. Acquisition of time-varying participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 812–15.

He, X. D., K. E. Torrance, F. X. Sillion, and D. P. Greenberg. 1991. A comprehensive physical model for light reflection. *Computer Graphics (SIGGRAPH '91 Proceedings), 25,* 175–86.

Heckbert, P. 1984. *The Mathematics of Quadric Surface Rendering and SOID,* 3-D Technical Memo, New York Institute of Technology Computer Graphics Lab.

Heckbert, P. 1989. Image Zooming Source Code. *www-2.cs.cmu.edu/~ph/src/zoom/*.

Heckbert, P. S. 1986. Survey of texture mapping. *IEEE Computer Graphics and Applications 6*(11), 56–67.

Heckbert, P. S. 1987. Ray tracing JELL-O brand gelatin. *Computer Graphics (SIGGRAPH '87 Proceedings), 21* 73–74.

Heckbert, P. S. 1989. Fundamentals of Texture Mapping and Image Warping. M.S. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Heckbert, P. 1990a. What are the coordinates of a pixel? In A. S. Glassner (Ed.), *Graphics Gems I,* pp. 246–48. San Diego: Academic Press.

Heckbert, P. S. 1990b. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '90), 24,* 145–54.

Heckbert, P. S., and P. Hanrahan. 1984. Beam tracing polygonal objects. In *Computer Graphics (Proceedings of SIGGRAPH '84), 18,* 119–27.

Heidrich, W., and H.-P. Seidel. 1998. Ray-tracing procedural displacement shaders. In *Graphics Interface '98,* pp. 8–16.

Heidrich, W., J. Kautz, P. Slusallek, and H.-P. Seidel. 1998. Canned lightsources. In *Rendering Techniques '98: Proceedings of the Eurographics Rendering Workshop*. 293–300.

Henyey, L. G., and J. L. Greenstein. 1941. Diffuse radiation in the galaxy. *Astrophysical Journal 93,* 70–83.

Hery, C. 2003. Implementing a skin BSSRDF. *SIGGRAPH 2003 RenderMan Course Notes.*

Herzog, R., and H.-P. Seidel. 2007. Lighting details preserving photon density estimation. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications,* 407–10.

Herzog, R., V. Havran, S. Kinuwaki, K. Myszkowski, and H.-P. Seidel. 2007. Global illumination using photon ray splatting. *Computer Graphics Forum (Proceedings of Eurographics 2007) 26*(3), 503–13.

Herzog, R., K. Myszkowski, and H.-P. Seidel. 2009. Anisotropic radiance-cache splatting for efficiently computing high-quality global illumination with lightcuts. *Computer Graphics Forum (Proceedings of Eurographics 2009) 28*(2). 259–68.

Hey, H., and W. Purgathofer. 2002. Importance sampling with hemispherical particle footprints. In *Proceedings of the 18th Spring Conference on Computer Graphics*.

Higham, N. 1986. Computing the polar decomposition—with applications. *SIAM Journal of Scientific and Statistical Computing, 7*(4), 1160–74.

Hoberock, J. 2008. Accelerating Physically-Based Light Transport Algorithms. Ph.D. thesis, University of Illinois at Urbana-Champaign.

Hoberock, J., V. Lu, Y. Jia, J. Hart. 2009. Stream compaction for deferred shading. *Proceedings of High Performance Graphics 2009,* pp. 173–80.

Hoffmann, C. M. 1989. *Geometric and Solid Modeling: An Introduction*. San Francisco: Morgan Kaufmann.

Hoppe, H., T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. 1994. Piecewise smooth surface reconstruction. In *Proceedings of SIGGRAPH '94,* Computer Graphics Proceedings, Annual Conference Series, Orlando, Florida, pp. 295–302.

Hunt, W. 2008. Corrections to the surface area metric with respect to mail-boxing. In *IEEE Symposium on Interactive Ray Tracing,* pp. 77–80.

Hunt, W., and B. Mark. 2008a. Ray-specialized acceleration structures for ray tracing. In *IEEE Symposium on Interactive Ray Tracing,* pp. 3–10.

Hunt, W., and B. Mark. 2008b. Adaptive acceleration structures in perspective space. In *IEEE Symposium on Interactive Ray Tracing,* pp. 117–17.

Hunt, W., W. Mark, and G. Stoll. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing,* pp. 81–88.

Hurley, J., A. Kapustin, A. Reshetov, and A. Soupikov. 2002. Fast ray tracing for modern general purpose CPU. In *Proceedings of GraphiCon 2002*.

Igarashi, T., K. Nishino and S. K. Nayar. 2007. The appearance of human skin: a survey. *Foundations and Trends in Computer Graphics and Vision 3*(1), 1–95.

Igehy, H. 1999. Tracing ray differentials. In *Proceedings of SIGGRAPH '99,* Computer Graphics Proceedings, Annual Conference Series, pp. 179–86.

Igehy, H., M. Eldridge, and K. Proudfoot. 1998. Prefetching in a texture cache architecture. In *1998 SIGGRAPH/Eurographics Workshop on Graphics Hardware,* pp. 133–42.

Igehy, H., M. Eldridge, and P. Hanrahan. 1999. Parallel texture caching. In *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware,* pp. 95–106.

Illuminating Engineering Society of North America Computer Committee. 1986. IES recommended standard file format for electronic transfer of photometric data. IES LM-63-1986.

Illuminating Engineering Society of North America. 2002. IESNA standard file format for electronic transfer of photometric data. BSR/IESNA Publication LM-63-2002. *www.iesna.org.*

Immel, D. S., M. F. Cohen, and D. P. Greenberg. 1986. A radiosity method for non-diffuse environments. In *Computer Graphics (SIGGRAPH '86 Proceedings),* Volume 20, pp. 133–42.

Institute of Electrical and Electronic Engineers. 1985. IEEE standard 754-1985 for binary floating-point arithmetic. Reprinted in *SIGPLAN 22*(2) 9–25.

Irawan, P. 2008. Appearance of Woven Cloth. Ph.D. thesis, Cornell University.

Ishimaru, A. 1978. *Wave Propagation and Scattering in Random Media*. Oxford: Oxford University Press.

Ize, T., P. Shirley, and S. Parker. 2007. Grid creation strategies for efficient ray tracing. In *IEEE Symposium on Interactive Ray Tracing,* pp. 27–32.

Ize, T., I. Wald, and S. Parker. 2008. Ray tracing with the BSP tree. In *IEEE Symposium on Interactive Ray Tracing,* pp. 159–166.

Jackson, W. H. 1910. The solution of an integral equation occurring in the theory of radiation. *Bulletin of the American Mathematical Society 16,* 473–75.

James, D., and K. Fatahalian. 2003. Precomputing interactive dynamic deformable scenes. *ACM Transactions on Graphics 22*(3), 879–87.

Jansen, F. W. 1986. Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. Lierop (Eds.), *Data Structures for Raster Graphics, Workshop Proceedings,* pp. 57–73. New York: Springer-Verlag.

Jarosz, W., M. Zwicker, and H. W. Jensen. 2008a. The beam radiance estimate for volumetric photon mapping. *Computer Graphics Forum (Proceedings of Eurographics 2008) 27*(2), 557–66.

Jarosz, W., M. Zwicker, and H. W. Jensen. 2008b. Irradiance gradients in the presence of participating media and occlusions. *Computer Graphics Forum 27*(4), 1087–96.

Jarosz, W., N. Carr, and H. Jensen. 2009. Importance sampling spherical harmonics. *Computer Graphics Forum (Proceedings of Eurographics 2009) 28*(2). 577–86.

Jensen, H. W. 1995. Importance driven path tracing using the photon map. In *Eurographics Rendering Workshop 1995,* pp. 326–35.

Jensen, H. W. 1996a. Global illumination using photon maps. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996,* pp. 21–30.

Jensen, H. W. 1996b. Rendering caustics on non-Lambertian surfaces. In *Graphics Interface '96,* pp. 116–21.

Jensen, H. W. 1997. Rendering caustics on non-Lambertian surfaces. *Computer Graphics Forum 16*(1), 57–64.

Jensen, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. Natick, Massachusetts: A. K. Peters.

Jensen, H. W., and J. Buhler. 2002. A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics 21*(3), 576–81.

Jensen, H. W., and N. Christensen. 1995. Optimizing path tracing using noise reduction filters. *Proceedings of WSCG,* 134–42.

Jensen, H. W., and P. H. Christensen. 1998. Efficient simulation of light transport in scenes with participating media using photon maps. In *SIGGRAPH '98 Conference Proceedings,* Annual Conference Series, pp. 311–20.

Jensen, H. W., J. Arvo, M. Fajardo, P. Hanrahan, D. Mitchell, M. Pharr, and P. Shirley. 2001a. State of the art in Monte Carlo ray tracing for realistic image synthesis. In *SIGGRAPH 2001 Course 29,* Los Angeles.

Jensen, H. W., S. R. Marschner, M. Levoy, and P. Hanrahan. 2001b. A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH 2001,* Computer Graphics Proceedings, Annual Conference Series, pp. 511–18.

Jensen, H. W., J. Arvo, P. Dutré, A. Keller, A. Owen, M. Pharr, and P. Shirley. 2003. Monte Carlo ray tracing. In *SIGGRAPH 2003 Course,* San Diego.

Jevans, D., and B. Wyvill. 1989. Adaptive voxel subdivision for ray tracing. In *Graphics Interface '89,* pp. 164–172.

Johnstone, M. S., and P. R. Wilson. 1999. The memory fragmentation problem: solved? *ACM SIGPLAN Notices 34*(3), 26–36.

Jones, T. 2005. Efficient generation of Poisson-disk sampling patterns. *Journal of Graphics Tools 11*(2), 27–36.

Kainz, F., R. Bogart, and D. Hess. 2002. OpenEXR Image File Format. *www.openexr.com*.

Kajiya, J. T. 1982. Ray tracing parametric patches. In *Computer Graphics (SIGGRAPH 1982 Conference Proceedings),* pp. 245–54.

Kajiya, J. T. 1983. New techniques for ray tracing procedurally defined objects. In *Computer Graphics (Proceedings of SIGGRAPH '83), 17,* 91–102.

Kajiya, J. T. 1985. Anisotropic reflection models. *Computer Graphics (Proceedings of SIGGRAPH '85), 19,* 15–21.

Kajiya, J. T. 1986. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings), 20,* 143–50.

Kajiya, J. T., and T. L. Kay. 1989. Rendering fur with three dimensional textures. *Computer Graphics (Proceedings of SIGGRAPH '89), 23,* 271–80.

Kajiya, J., and M. Ullner. 1981. Filtering high quality text for display on raster scan devices. In *Computer Graphics (Proceedings of SIGGRAPH '81),* pp. 7–15.

Kajiya, J. T., and B. P. Von Herzen. 1984. Ray tracing volume densities. In *Computer Graphics (Proceedings of SIGGRAPH '84),* Volume 18, pp. 165–74.

Kalos, M. H., and P. A. Whitlock. 1986. *Monte Carlo Methods: Volume I: Basics*. New York: Wiley.

Kalra, D., and A. H. Barr. 1989. Guaranteed ray intersections with implicit surfaces. In *Computer Graphics (Proceedings of SIGGRAPH '89),* Volume 23, pp. 297–306.

Kammaje, R. and B. Mora. 2007. A study of restricted BSP trees for ray tracing. In *IEEE Symposium on Interactive Ray Tracing,* pp. 55–62.

Kaplan, M. R. 1985. The uses of spatial coherence in ray tracing. In *ACM SIGGRAPH Course Notes 11.*

Kautz, J., and M. McCool. 1999. Interactive rendering with arbitrary BRDFs using separable approximations. In *Proceedings of the 10th Eurographics Workshop on Rendering*. 281–92.

Kautz, J., P.-P. Sloan, and J. Snyder. 2002. Fast, arbitrary BRDF shading for low-frequency lighting using spherical harmonics. In *13th Eurographics Workshop on Rendering*. 301–08.

Kay, D. S., and D. P. Greenberg. 1979. Transparency for computer synthesized images. In *Computer Graphics (SIGGRAPH '79 Proceedings),* Volume 13, pp. 158–64.

Kay, T., and J. Kajiya. 1986. Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings),* Volume 20, pp. 269–78.

Kelemen, C., L. Szirmay-Kalos, G. Antal, and F. Csonka. 2002. A simple and robust mutation strategy for the Metropolis light transport algorithm. *Computer Graphics Forum 21*(3), 531–40.

Keller, A. 1996. Quasi-Monte Carlo radiosity. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996,* pp. 101–10.

Keller, A. 1997. Instant radiosity. In *Proceedings of SIGGRAPH '97,* Computer Graphics Proceedings, Annual Conference Series, Los Angeles, pp. 49–56.

Keller, A. 1998. Quasi-Monte Carlo Methods for Photorealistic Image Synthesis. Ph.D. thesis, Shaker Verlag Aachen.

Keller, A. 2001. *Strictly Deterministic Sampling Methods in Computer Graphics*. *mental images Technical Report.* Also in *SIGGRAPH 2003 Monte Carlo Course Notes.*

Keller, A., and I. Wald. 2000. Efficient importance sampling techniques for the photon map. In *Proceedings of Vision, Modeling and Visualization 2000,* pp. 271–79.

Keller, A. 2006. Myths of computer graphics. In *Monte Carlo and Quasi-Monte Carlo Methods 2004,* Springer Berlin Heidelberg. 217–43.

Kensler, A., and P. Shirley. 2006. Optimizing ray-triangle intersection via automated search. In *IEEE Symposium on Interactive Ray Tracing,* pp. 33–38.

Kensler, A. 2008. Tree rotations for improving bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing,* pp. 73–76.

Kensler, A., A. Knoll and P. Shirley. 2008. Better Gradient Noise. *Technical Report UUSCI-2008-001, SCI Institute, University of Utah*.

King, L. V. 1913. On the scattering and absorption of light in gaseous media, with applications to the intensity of sky radiation. *Philosophical Transactions of the Royal Society of London. Series A. Mathematical and Physical Sciences 212,* 375–433.

Kirk, D., and J. Arvo. 1988. The ray tracing kernel. In *Proceedings of Ausgraph '88,* pp. 75–82.

Kirk, D. B., and J. Arvo. 1991. Unbiased sampling techniques for image synthesis. *Computer Graphics (SIGGRAPH '91 Proceedings),* Volume 25, pp. 153–56.

Klassen, R. V. 1987. Modeling the effect of the atmosphere on light. *ACM Transactions on Graphics 6*(3), 215–37.

Klimaszewski, K. S., and T. W. Sederberg. 1997. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications 17*(1), 42–51.

Knoll, A., Y. Hijazi, C. D. Hansen, I. Wald, and H. Hagen. 2009. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum 28*(1), 26–40.

Knuth, D. E. 1984. Literate programming. *The Computer Journal 27,* 97–111. Reprinted in D. E. Knuth, *Literate Programming,* Stanford Center for the Study of Language and Information, 1992.

Knuth, D. E. 1986. *MetaFont: The Program*. Reading, Massachusetts: Addison-Wesley.

Knuth, D. E. 1993a. *TEX: The Program*. Reading, Massachusetts: Addison-Wesley.

Knuth, D. E. 1993b. *The Stanford GraphBase*. New York: ACM Press and Addison-Wesley.

Kolb, C., D. Mitchell, and P. Hanrahan. 1995. A realistic camera model for computer graphics. *SIGGRAPH '95 Conference Proceedings,* Annual Conference Series, pp. 317–24.

Kollig, T., and A. Keller. 2000. Efficient bidirectional path tracing by randomized quasi-Monte Carlo integration. In *Monte Carlo and Quasi-Monte Carlo Methods 2000,* pp. 290–305. Berlin: Springer-Verlag.

Kollig, T., and A. Keller. 2002. Efficient multidimensional sampling. *Computer Graphics Forum (Proceedings of Eurographics 2002),* Volume 21, pp. 557–63.

Kollig, T., and A. Keller. 2003. Efficient illumination by high dynamic range images. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering,* pp. 45–51.

Kollig. T., and A. Keller. 2004. Illumination in the presence of weak singularities. In *Proceedings of Monte Carlo and Quasi-Monte Carlo Methods*.

Kontkanen, J. and T. Aila. 2006. Ambient occlusion for animated characters. In *Rendering Techniques 2006 (Proc. Eurographics Symposium on Rendering),* 343–48.

Kontkanen, J. and S. Laine. 2005. Ambient occlusion fields. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games,* 41–48.

Kontkanen, J. and S. Laine. 2006. Sampling precomputed volumetric lighting. *Journal of Graphics, GPU, and Game Tools 11*(3), 1–16.

Kontkanen, J., J. Räsänen, and A. Keller. 2004. Irradiance filtering for monte carlo ray tracing. *Monte Carlo and Quasi-Monte Carlo Methods,* 259–72.

Kristensen, A., T. Akenine-Möller, and H. W. Jensen. 2005. Precomputed local radiance transfer for real-time lighting design. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1208–15.

Křivánek, J., and P. Gautron. 2009. *Practical Global Illumination with Irradiance Caching*. Morgan and Claypool Publishers.

Křivánek, J., P. Gautron, K. Bouatouch, and S. Pattanaik. 2005a. Improved radiance gradient computation. In *Spring Conference on Computer Graphics,* 155–59.

Křivánek, J., P. Gautron, S. Pattanaik, and K. Bouatouch. 2005b. Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics 11*(5). 550–61.

Křivánek, J., P. Gautron, G. Ward, O. Arikan, H. W. Jensen. 2007. Practical global illumination with irradiance caching. *ACM SIGGRAPH Course Notes 16.*

Křivánek, J., J. Konttinen, S. Pattanaik, K. Bouatouch, and J. Zara. 2006. Fast approximation to spherical harmonic rotation. In *Proceedings of 22nd Spring Conference on Computer Graphics (SCCG2006).*

Lacewell, D., B. Burley, S. Boulos, and P. Shirley. 2008. Raytracing prefiltered occlusion for aggregate geometry. In *IEEE Symposium on Interactive Ray Tracing,* pp. 19–26.

Lafortune, E. 1996. Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering. Ph. D. thesis, Katholieke Universiteit Leuven.

Lafortune, E., and Y. Willems. 1994. A theoretical framework for physically based rendering. *Computer Graphics Forum 13*(2), 97–107.

Lafortune, E. P., and Y. D. Willems. 1996. Rendering participating media with bidirectional path tracing. In *Eurographics Rendering Workshop 1996,* pp. 91–100.

Lafortune, E. P. F., S.-C. Foo, K. E. Torrance, and D. P. Greenberg. 1997. Non-linear approximation of reflectance functions. In *Proceedings of SIGGRAPH '97,* Computer Graphics Proceedings, Annual Conference Series, Los Angeles, pp. 117–26.

Lagae, A. and P. Dutré. 2005. An efficient ray-quadrilateral intersection test. *Journal of Graphics Tools 10*(4), 23–32.

Lagae, A. and P. Dutré. 2008a. Compact, fast, and robust grids for ray tracing. In *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering) 27*(4), 1235–1244.

Lagae, A. and P. Dutré. 2008b. Accelerating ray tracing using constrained tetrahedralizations. In *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering) 27*(4), 1303–12.

Lagae, E., and P. Dutré. 2008c. A comparison of methods for generating Poisson disk distributions. *Computer Graphics Forum 27*(1), 114–29.

Lagae, A., S. Lefebvre, G. Drettakis and P. Dutré. 2009. Procedural noise using sparse Gabor convolution. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009) 28*(3), 54:1–54:10.

Lam, M. S., E. E. Rothberg, and M. E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV),* Palo Alto, California.

Lambert, J. H. 1760. *Photometry, or, On the Measure and Gradations of Light, Colors, and Shade*. The Illuminating Engineering Society of North America. Translated by David L. DiLaura in 2001.

Lang, S. 1986. *An Introduction to Linear Algebra*. New York: Springer-Verlag.

Langlands, A., and T. Mertens. 2007. Noise-free BSSRDF rendering on the cheap. *SIGGRAPH 2007 posters*.

Lansdale, R. C. 1991. Texture Mapping and Resampling for Computer GLraphics. M.S. thesis, Department of Electrical Engineering, University of Toronto.

Larson, G. W. 1998. LogLUV encoding for full-gamut, high-dynamic range images. *Journal of Graphics Tools 3*(1), 15–31.

Larson, G. W., and R. A. Shakespeare. 1998. *Rendering with Radiance: The Art and Science of Lighting Visualization*. San Francisco: Morgan Kaufmann.

Lauterbach, C., M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast BVH construction on GPUs. *Computer Graphics Forum (Eurographics 2009 Conference Proceedings) 28*(2).

Lawrence, J., S. Rusinkiewicz, and R. Ramamoorthi. 2004. Efficient BRDF importance sampling using a factored representation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004) 23*(3), 496–505.

Lawrence, J., S. Rusinkiewicz, and R. Ramamoorthi. 2005. Adaptive numerical cumulative distribution functions for efficient importance sampling. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering,* 11–20.

Lee, M. E., R. A. Redner, and S. P. Uselton. 1985. Statistically optimized sampling for distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH '85),* Volume 19, pp. 61–67.

Lee, M., and R. Redner. 1990. A note on the use of nonlinear filtering in computer graphics. *IEEE Computer Graphics and Applications 10*(3), 23–29.

Legakis, J. 1998. Fast multi-layer fog. *SIGGRAPH '98 Sketch.*

Lehtinen, J. 2004. Foundations of Precomputed Radiance Transfer. M.S. thesis, Helsinki University of Technology.

Lehtinen, J. 2007. A framework for precomputed and captured light transport. *ACM Transactions on Graphics 26*(4), 13:1–13:22.

Lehtinen, J. and J. Kautz. 2003. Matrix radiance transfer. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics,* 59–64.

Lehtinen, J., M. Zwicker, E. Turquin, J. Kontkanen, F. Durand, F. Sillion, and T. Aila. 2008. A meshless hierarchical representation for light transport. *ACM Transactions on Graphics 27*(3), 1–9.

Levine, J. R., T. Mason, and D. Brown. 1992. *lex & yacc*. O'Reilly & Associates.

Levoy, M. 1988. Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*(3), 29–37.

Levoy, M. 1990a. Efficient ray tracing of volume data. *ACM Transactions on Graphics 9*(3), 245–61.

Levoy, M. 1990b. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications 10*(2), 33–40.

Levoy, M., and P. M. Hanrahan. 1996. Light field rendering. In *Proceedings of SIGGRAPH '96,* Computer Graphics Proceedings, Annual Conference Series, pp. 31–42.

Levoy, M., and T. Whitted. 1985. *The Use of Points as a Display Primitive,* Technical Report 85-022. Computer Science Department, University of North Carolina at Chapel Hill.

Lewis, J.-P. 1989. Algorithms for solid noise synthesis. In *Computer Graphics (Proceedings of SIGGRAPH '89),* Volume 23, pp. 263–70.

Lext, J., U. Assarsson, and T. Möller. 2001. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications 21*(2), 22–30.

Li, K., F. Pellacini, and K. Torrance. 2005. A hybrid Monte Carlo method for accurate and efficient subsurface scattering. In *Rendering Techniques (Proceedings of the Eurographics Rendering Symposium,* 283–90.

Li, X., W. Wang, R. R. Martin, and A. Bowyer. 2003. Using low-discrepancy sequences and the Crofton formula to compute surface areas of geometric models. *Computer Aided Design 35*(9), 771–82.

Liu, J. S. 2001. *Monte Carlo Strategies in Scientific Computing*. New York: Springer-Verlag.

Liu, X., P.-P. Sloan, H.-Y. Shum, and J. Snyder. 2004. All-frequency precomputed radiance transfer for glossy objects. In *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering 2004),* 337–44.

Logie, J. R., and J. W. Patterson. 1994. Inverse displacement mapping in the general case. *Computer Graphics Forum 14*(5), 261–73.

Lokovic, T., and E. Veach. 2000. Deep shadow maps. In *Proceedings of ACM SIGGRAPH 2000,* Computer Graphics Proceedings, Annual Conference Series, pp. 385–92.

Lommel, E. 1889. Die Photometrie der diffusen Zurückwerfung. *Annalen der Physik 36,* 473–502.

Loop, C. 1987. Smooth Subdivision Surfaces Based on Triangles. M.S. thesis, University of Utah.

Lu, R., J. J. Koenderink, and A. M. L. Kappers. 1999. Specularities on surfaces with tangential hairs or grooves. *Computer Vision and Image Understanding 78,* 320–35.

Lukaszewski, A. 2001. Exploiting coherence of shadow rays. In *AFRIGRAPH 2001,* pp. 147–150. ACM SIGGRAPH.

MacDonald, J. D., and K. S. Booth. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer 6*(3), 153–66.

Machiraju, R., and R. Yagel. 1996. Reconstruction error characterization and control: A sampling theory approach. *IEEE Transactions on Visualization and Computer Graphics 2*(4).

Mahajan, D., Y.-T. Tseng, and R. Ramamoorthi. 2008. An analysis of the in-out BRDF factorization for view-dependent relighting *Computer Graphics Forum (Proceedings of Eurographics) 27*(4), 1137–45.

Mahovsky, J. 2005. Ray Tracing with Reduced-Precision Bounding Volume Hierarchies. Ph.D. thesis, University of Calgary.

Malacara, D. 2002. *Color Vision and Colorimetry: Theory and Applications*. SPIE—The International Society for Optical Engineering.

Mann, S., N. Litke, and T. DeRose. 1997. A Coordinate Free Geometry ADT. Research Report CS-97-15, Computer Science Department, University of Waterloo. Available at *ftp://cs-archive.uwaterloo.ca/cs-archive/CS-97-15/.*

Mansson, E., J. Munkberg, and T. Akenine-Möller. 2007. Deep coherent ray tracing. *Proceedings of IEEE Symposium on Interactive Ray Tracing,* 79–85.

Marschner, S. 1998. Inverse Rendering for Computer Graphics. Ph.D. thesis, Cornell University.

Marschner, S. R., H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan. 2003. Light scattering from human hair fibers. *ACM Transactions on Graphics 22*(3), 780–91.

Marschner, S. R., and R. J. Lobb. 1994. An evaluation of reconstruction filters for volume rendering. In *Proceedings of Visualization '94,* Washington, D.C., pp. 100–107.

Marschner, S., S. Westin, A. Arbree, and J. Moon. 2005. Measuring and modeling the appearance of finished wood. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 727–34.

Marschner, S. R., S. H. Westin, E. P. F. Lafortune, K. E. Torrance, and D. P. Greenberg. 1999. Image-based BRDF measurement including human skin. In *Eurographics Rendering Workshop 1999,* Granada, Spain: Springer Wien/Eurographics.

Martin, W., E. Cohen, R. Fish, and P. S. Shirley. 2000. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools 5*(1), 27–52.

Matusik, W., H. Pfister, M. Brand, and L. McMillan. 2003a. Efficient isotropic BRDF measurement. *Proceedings of the 14th Eurographics Workshop on Rendering*. 241–47.

Matusik, W., H. Pfister, M. Brand, and L. McMillan. 2003b. A data-driven reflectance model. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003) 22*(3), 759–69.

Max, N. L. 1986. Atmospheric illumination and shadows. In *Computer Graphics (Proceedings of SIGGRAPH '86),* Volume 20, pp. 117–24.

Max, N. L. 1988. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer 4*(2), 109–17.

Max, N. L. 1995. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*(2), 99–108.

McCluney, W. R. 1994. *Introduction to Radiometry and Photometry*. Boston: Artech House.

McCormack, J., R. Perry, K. I. Farkas, and N. P. Jouppi. 1999. Feline: fast elliptical lines for anisotropic texture mapping. In *Proceedings of SIGGRAPH '99,* Computer Graphics Proceedings, Annual Conference Series, Los Angeles, pp. 243–250.

Meijering, E. 2002. A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proceedings of the IEEE 90*(3), 319–42.

Meijering, E. H. W., W. J. Niessen, J. P. W. Pluim, and M. A. Viergever. 1999. Quantitative comparison of sinc-approximating kernels for medical image interpolation. In C. Taylor and A. Colchester (Eds.), *Medical Image Computing and Computer-Assisted Intervention—MICCAI 1999,* pp. 210–17. Berlin: Springer-Verlag.

Méndez-Feliu, À., and M. Sbert. 2009. From obscurances to ambient occlusion: a survey. *The Visual Computer 25*(2), 181–96.

Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. 1953. Equation of state calculations by fast computing machines. *Journal of Chemical Physics 21*(6), 1087–92.

Meyer, G. W., and D. P. Greenberg. 1980. Perceptual color spaces for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH '80),* Volume 14, Seattle, Washington, pp. 254–261.

Meyer, G. W., H. E. Rushmeier, M. F. Cohen, D. P. Greenberg, and K. E. Torrance. 1986. An experimental evaluation of computer graphics imagery. *ACM Transactions on Graphics 5*(1), 30–50.

Mikkelsen, M. 2008. Simulation of Wrinkled Surfaces Revisited. M.S. thesis, University of Copenhagen.

Miller, G. S., and C. R. Hoffman. 1984. Illumination and reflection maps: simulated objects in simulated and real environments. *Course Notes for Advanced Computer Graphics Animation, SIGGRAPH '84.*

Miller, G. 1994. Efficient algorithms for local and global accessibility shading. In *Proceedings of SIGGRAPH '94,* Computer Graphics Proceedings, Annual Conference Series, pp. 319–26.

Mitchell, D. P. 1987. Generating antialiased images at low sampling densities. *Computer Graphics (SIGGRAPH '87 Proceedings),* Volume 21, pp. 65–72.

Mitchell, D. P. 1990. Robust ray intersection with interval arithmetic. In *Graphics Interface '90,* pp. 68–74.

Mitchell, D. P. 1991. Spectrally optimal sampling for distributed ray tracing. *Computer Graphics (SIGGRAPH '91 Proceedings),* Volume 25, pp. 157–64.

Mitchell, D. P. 1992. Ray tracing and irregularities of distribution. In *Third Eurographics Workshop on Rendering,* Bristol, United Kingdom, pp. 61–69.

Mitchell, D. P. 1996a. Software interface for sampling routines. Personal communication.

Mitchell, D. P. 1996b. Consequences of stratified sampling in graphics. In *Proceedings of SIGGRAPH '96,* Computer Graphics Proceedings, Annual Conference Series, New Orleans, Louisiana, pp. 277–80.

Mitchell, D. P., and P. Hanrahan. 1992. Illumination from curved reflectors. In *Computer Graphics (Proceedings of SIGGRAPH '92),* Volume 26, pp. 283–91.

Mitchell, D. P., and A. N. Netravali. 1988. Reconstruction filters in computer graphics. *Computer Graphics (SIGGRAPH '88 Proceedings),* Volume 22, pp. 221–28.

Möller, T., R. Machiraju, K. Mueller, and R. Yagel. 1997. Evaluation and design of filters using a Taylor series expansion. *IEEE Transactions on Visualization and Computer Graphics 3*(2), 184–99.

Möller, T., and B. Trumbore. 1997. Fast, minimum storage ray–triangle intersection. *Journal of Graphics Tools 2*(1), 21–28.

Moon, J., and S. Marschner. 2006. Simulating multiple scattering in hair using a photon mapping approach. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) 25*(3), 1067–74.

Moon, J., B. Walter, and S. Marschner. 2007. Rendering discrete random media using precomputed scattering solutions. *Rendering Techniques 2007: 18th Eurographics Workshop on Rendering,* 231–42.

Moon, J., B. Walter, and S. Marschner. 2008. Efficient multiple scattering in hair using spherical harmonics. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27*(3). 31:1–31:7.

Moon, P., and D. E. Spencer. 1936. *The Scientific Basis of Illuminating Engineering*. New York: McGraw-Hill.

Moon, P., and D. E. Spencer 1948. *Lighting Design*. Reading, Masschusetts: Addison-Wesley.

Moore, R. E. 1966. *Interval Analysis*. Englewood Cliffs, New Jersey: Prentice Hall.

Moravec, H. 1981. 3D graphics and the wave theory. In *Computer Graphics,* Volume 15, pp. 289–96.

Motwani, R., and P. Raghavan. 1995. *Randomized Algorithms*. Cambridge, U.K.: Cambridge University Press.

Müller, K., T. Techmann, and D. Fellner. 2003. Adaptive ray tracing of subdivision surfaces. *Computer Graphics Forum 22*(3), 553–62.

Müller, G., J. Meseth, M. Sattler, R. Sarlette and R. Klein. 2005. Acquisition, synthesis and rendering of bidirectional texture functions. *Computer Graphics Forum (Eurographics State of the Art Report) 24*(1), 83–109.

Musgrave, K. 1992. A panoramic virtual screen for ray tracing. In D. Kirk (Ed.), *Graphics Gems III,* pp. 288–94. San Diego: Academic Press.

Narasimhan, S., M. Gupta, C. Donner, R. Ramamoorthi, S. Nayar, and H. W. Jensen. 2006. Acquiring scattering properties of participating media by dilution. *ACM Transactions on Graphics 25*(3), 1003–12.

Nayar, S. K., K. Ikeuchi, and T. Kanade. 1991. Surface reflection: physical and geometrical perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence 17*(7), 611–34.

Naylor, B. 1993. Constructing good partition trees. In *Graphics Interface '93,* pp. 181–91.

Neulander, I. 2009. Smoother Subsurface Scattering. *SIGGRAPH Talks*.

Neyret, F. 1996. Synthesizing verdant landscapes using volumetric textures. In *Eurographics Rendering Workshop 1996,* pp. 215–24.

Neyret, F. 1998. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics 4*(1), 55–70.

Nicodemus, F., J. Richmond, J. Hsia, I. Ginsburg, and T. Limperis. 1977. *Geometrical Considerations and Nomenclature for RLeflectance*. NBS Monograph 160, Washington, D. C.: National Bureau of Standards, U.S. Department of Commerce.

Nichols, G., and C. Wyman. 2009. Multiresolution splatting for indirect illumination. *ACM Symposium on Interactive 3D Graphics and Games,* 83–90.

Niederreiter, H. 1992. *Random Number Generation and Quasi–Monte Carlo Methods*. Philadelphia: Society for Industrial and Applied Mathematics.

Nijasure, M., S. Pattanaik, V. Goel. 2005. Real-time global illumination on GPU. *Journal of Graphics Tools 10*(2). 55–71.

Nimeroff, J., E. Simoncelli, and J. Dorsey. 1994. Efficient re-rendering of naturally illuminated environments. In *Fifth Eurographics Workshop on Rendering,* 359–73.

Nishita, T., and E. Nakamae. 1986. Continuous tone representation of three-dimensional objects illuminated by sky light. In *Computer Graphics (Proceedings of SIGGRAPH '86),* Volume 20, pp. 125–32.

Nishita, T., Y. Miyawaki, and E. Nakamae. 1987. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. In *Computer Graphics (Proceedings of SIGGRAPH '87),* Volume 21, pp. 303–10.

Ngan, A., F. Durand, and W. Matusik. 2005. Experimental analysis of BRDF models. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering,* 117–26.

Ng, R., R. Ramamoorthi, and P. Hanrahan. 2003. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003) 22*(3), 376–81.

Ng, R., R. Ramamoorthi, and P. Hanrahan. 2004. Triple product wavelet integrals for all-frequency relighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004) 23*(3), 477–87.

Norton, A., A. P. Rockwood, and P. T. Skolmoski. 1982. Clamping: a method of antialiasing textured surfaces by bandwidth limiting in object space. In *Computer Graphics (Proceedings of SIGGRAPH '82),* Volume 16, pp. 1–8.

Ökten, G., and M. Shah. 2008. *Random and Deterministic Digit-Scrambling of the Halton Sequence,* unpublished manuscript.

Ooi, B. C., K. McDonell, and R. Sacks-Davis. 1987. Spatial kd-tree: a data structure for geographic databases. In *Proceedings of the IEEE COMPSAC Conference*.

Oren, M., and S. K. Nayar. 1994. Generalization of Lambert's reflectance model. *Proceedings of SIGGRAPH '94, Computer Graphics Proceedings, Annual Conference Series,* pp. 239–46. New York: ACM Press.

Overbeck, R., C. Donner, and R. Ramamoorthi. 2009. Adaptive wavelet rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009) 28*(5).

Owen, A. B. 1998. Latin supercube sampling for very high-dimensional simulations. *Modeling and Computer Simulation 8*(1), 71–102.

Owen, A., and S. Tribble. 2005. A quasi-Monte Carlo Metropolis algorithm. *Proceedings of the National Academy of Sciences 102*(25), 8844–49.

Parker, S., S. Boulos, J. Bigler, and A. Robison. 2007. RTSL: a ray tracing shading language. *Proceedings of IEEE Symposium on Interactive Ray Tracing*.

Parker, S., W. Martin, P.-P. J. Sloan, P. S. Shirley, B. Smits, and C. Hansen. 1999. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics,* pp. 119–26.

Patmore, C. 1993. Illumination of dense foliage models. In *Fourth Eurographics Workshop on Rendering,* pp. 63–72.

Patney, A., M. S. Ebeida, and J. D. Owens. 2009. Parallel view-dependent tessellation of Catmull–Clark subdivision surfaces. In *Proceedings of High Performance Graphics,* 99–108.

Pattanaik, S. N., and S. P. Mudur. 1995. Adjoint equations and random walks for illumination computation. *ACM Transactions on Graphics 14*(1), 77–102.

Patterson, D., and J. Hennessy. 2006. *Computer Architecture: A Quantitative Approach.* San Francisco: Morgan Kaufmann.

Patterson, J. W., S. G. Hoggar, and J. R. Logie. 1991. Inverse displacement mapping. *Computer Graphics Forum 10*(2), 129–39.

Pauly, M. 1999. Robust Monte Carlo Methods for Photorealistic Rendering of Volumetric Effects. Master's thesis, Universität Kaiserslautern.

Pauly, M., T. Kollig, and A. Keller. 2000. Metropolis light transport for participating media. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering,* pp. 11–22.

Peachey, D. R. 1985. Solid texturing of complex surfaces. *Computer Graphics (SIGGRAPH '85 Proceedings),* Volume 19, pp. 279–86.

Peachey, D. R. 1990. Texture on demand. Pixar Technical Memo #217. Available from *http://graphics.pixar.com/library/TOD/*.

Pearce, A. 1991. A recursive shadow voxel cache for ray tracing. In J. Arvo (Ed.), *Graphics Gems II,* pp. 273–74. San Diego: Academic Press.

Peercy, M. S. 1993. Linear color representations for full spectral rendering. *Computer Graphics (SIGGRAPH '93 Proceedings),* Volume 27, pp. 191–98.

Peercy, M. S., M. Olano, J. Airey, and P. J. Ungar. 2000. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000,* Computer Graphics Proceedings, Annual Conference Series, pp. 425–32.

Peers, P., K. vom Berge, W. Matusik, R. Ramamoorthi, J. Lawrence, S. Rusinkiewicz, and P. Dutré. 2006. A compact factored representation of heterogeneous subsurface scattering. *ACM Transactions on Graphics 25*(3), 746–53.

Pegoraro, V., and S. Parker. 2009. An analytical solution to single scattering in homogeneous participating media. *Computer Graphics Forum (Proceedings of Eurographics) 28*(2), 329–35.

Pegoraro, V., M. Schott, and S. Parker. 2009. An analytical approach to single scattering for anisotropic media and light distributions. In *Proceedings of Graphics Interface 2009,* 71–77.

Pegoraro, V., C. Brownlee, P. Shirley, and S. Parker. 2008a. Towards interactive global illumination effects via sequential Monte Carlo adaptation. *IEEE Symposium on Interactive Ray Tracing,* pp. 107–14.

Pegoraro, V., I. Wald, and S. Parker. 2008b. Sequential Monte Carlo adaptation in low-anisotropy participating media. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering 27*(4), 1097–1104.

Pellacini, F., K. Vidimče, A. Lefohn, A. Mohr, M. Leone, and J. Warren. 2005. LPICS: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics 24*(3), 464–70.

Perlin, K. 1985a. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings),* Volume 19, pp. 287–96.

Perlin, K. 1985b. State of the art in image synthesis. *SIGGRAPH Course Notes 11.*

Perlin, K. 2002. Improving noise. *ACM Transactions on Graphics 21*(3), 681–82.

Perlin, K., and E. M. Hoffert. 1989. Hypertexture. In *Computer Graphics (Proceedings of SIGGRAPH '89),* Volume 23, pp. 253–62.

Peter, I., and G. Pietrek. 1998. Importance driven construction of photon maps. In *Eurographics Rendering Workshop 1998,* pp. 269–80.

Pfister, H., M. Zwicker, J. van Baar, and M. Gross. 2000. Surfels: Surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH 2000,* Computer Graphics Proceedings, Annual Conference Series, pp. 335–42.

Pharr, M., and P. Hanrahan. 1996. Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996,* pp. 31–40.

Pharr, M., and P. M. Hanrahan. 2000. Monte Carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proceedings of ACM SIGGRAPH 2000,* Computer Graphics Proceedings, Annual Conference Series, pp. 75–84.

Pharr, M., C. Kolb, R. Gershbein, and P. M. Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH '97,* Computer Graphics Proceedings, Annual Conference Series, pp. 101–08.

Phong, B.-T. 1975. Illumination for computer generated pictures. *Communications of the ACM 18*(6), 311–17.

Phong, B.-T., and F. C. Crow. 1975. Improved rendition of polygonal models of curved surfaces. In *Proceedings of the 2nd USA–Japan Computer Conference*.

Pixar Animation Studios. 2000. The RenderMan Interface. Version 3.2.

Popov, S., J. Gunther, H. P. Seidel, and P. Slusallek. 2006. Experinces with streaming construction of SAH kd-trees. In *IEEE Symposium on Interactive Ray Tracing,* pp. 89–94.

Popov, S., R. Dimov, I. Georgiev, and P. Slusallek. 2009. Object partitioning considered harmful: space subdivision for BVHs. *Proceedings of High Performance Graphics 2009,* pp. 15–22.

Porumbescu, S., B. Budge, L. Feng, and K. Joy. 2005. Shell maps. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 626–33.

Potmesil, M., and I. Chakravarty. 1981. A lens and aperture camera model for synthetic image generation. In *Computer Graphics (Proceedings of SIGGRAPH '81),* Volume 15, pp. 297–305.

Potmesil, M., and I. Chakravarty. 1982. Synthetic image generation with a lens and aperture camera model. *ACM Transactions on Graphics 1*(2), 85–108.

Potmesil, M., and I. Chakravarty. 1983. Modeling motion blur in computer-generated images. In *Computer Graphics (Proceedings of SIGGRAPH 83),* Volume 17, Detroit, Michigan, pp. 389–99.

Poulin, P., and A. Fournier. 1990. A model for anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH '90),* Volume 24, pp. 273–82.

Poynton, C. 2002a. Frequently-Asked Questions About Color. *www.poynton.com/ColorFAQ .html.*

Poynton, C. 2002b. Frequently-Asked Questions About Gamma. *www.poynton.com/ GammaFAQ.html*

Preetham, A. J., P. S. Shirley, and B. E. Smits. 1999. A practical analytic model for daylight. In *Proceedings of SIGGRAPH '99,* Computer Graphics Proceedings, Annual Conference Series, pp. 91–100.

Preisendorfer, R. W. 1965. *Radiative Transfer on Discrete Spaces*. Oxford: Pergamon Press.

Preisendorfer, R. W. 1976. *Hydrologic Optics*. Honolulu, Hawaii: U.S. Department of Commerce, National Oceanic and Atmospheric Administration.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 1992. *Numerical Recipes in C: The Art of Scientific Computing* (2nd ed.). Cambridge: Cambridge University Press.

Prusinkiewicz, P. 1986. Graphical applications of L-systems. In *Graphics Interface '86,* pp. 247–53.

Prusinkiewicz, P., M. James, and R. Mech. 1994. Synthetic topiary. In *Proceedings of SIGGRAPH '94,* Computer Graphics Proceedings, Annual Conference Series, pp. 351–58.

Prusinkiewicz, P., L. Mündermann, R. Karwowski, and B. Lane. 2001. The use of positional information in the modeling of plants. In *Proceedings of ACM SIGGRAPH 2001,* Computer Graphics Proceedings, Annual Conference Series, pp. 289–300.

Purcell, T. J., I. Buck, W. R. Mark, and P. Hanrahan. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*(3), 703–12.

Purcell, T. J., C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. 2003. Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003,* pp. 41–50.

Purgathofer, W. 1987. A statistical mothod for adaptive stochastic sampling. *Computers & Graphics 11*(2), 157–62.

Ragan-Kelley, J., C. Kilpatrick, B. Smith, D. Epps, P. Green, C. Hery, and F. Durand. 2007. The Lightspeed automatic interactive lighting preview system. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26*(3), 25:1–25:11.

Ramamoorthi. R. 2002. A Signal-Processing Framework for Forward and Inverse Rendering. Ph.D. thesis, Stanford University.

Ramamoorthi, R. 2007. Precomputation-based rendering. In *Foundations and Trends in Computer Graphics and Vision 3*(4), 281–369.

Ramamoorthi, R., and A. Barr. 1997. Fast construction of accurate quaternion splines. In *Proceedings of SIGGRAPH '97,* Computer Graphics Proceedings, Annual Conference Series, Los Angeles, pp. 287–92.

Ramamoorthi. R., and P. Hanrahan. 2001a. An efficient representation for irradiance environment maps. In *Proceedings of ACM SIGGRAPH 2001,* Computer Graphics Proceedings, Annual Conference Series, pp. 497–500.

Ramamoorthi. R., and P. Hanrahan. 2001b. On the relationship between radiance and irradiance: determining the illumination from images of a convex Lambertian object. *Journal of the Optical Society of America (JOSA A)* 10, 2448–59.

Ramamoorthi. R., and P. Hanrahan. 2004. A signal-processing framework for reflection. *ACM Transactions on Graphics 23*(4), 1004–42.

Ramasubramanian, M., S. N. Pattanaik, and D. P. Greenberg. 1999. A perceptually based physical error metric for realistic image synthesis. In *Proceedings of SIGGRAPH '99,* Computer Graphics Proceedings, Annual Conference Series, Los Angeles, pp. 73–82.

Raso, M., and A. Fournier. 1991. A piecewise polynomial approach to shading using spectral distributions. In *Graphics Interface '91,* pp. 40–46. Canadian Information Processing Society.

Reeves, W. T., D. H. Salesin, and R. L. Cook. 1987. Rendering antialiased shadows with depth maps. In *Computer Graphics (Proceedings of SIGGRAPH '87),* Volume 21, pp. 283–91.

Reichert, M. C. 1992. A Two-Pass Radiosity Method Driven by Lights and Viewer Position. Master's thesis, Cornell University.

Reinhard, E., G. Ward, S. Pattanaik, and P. Debevec. 2005. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann.

Ren, Z., R. Wang, J. Snyder, K. Zhou, X. Liu, B. Sun, P.-P. Sloan, H. Bao, Q. Peng, and B. Guo. 2006. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. *ACM Transactions on Graphics 25*(3), 977–86.

Ren, Z., K. Zhou, S. Lin, and B. Guo. 2008. Gradient-based interpolation and sampling for real-time rendering of inhomogeneous, single-scattering media. *Computer Graphics Forum (Proceedings of Pacific Graphics 2008) 27*(7), 1945–53.

Reshetov, A., A. Soupikov, and J. Hurley. 2005. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1176–85.

Reshetov, A. 2007. Faster ray packets–triangle intersection through vertex culling. In *Proceedings of IEEE Symposium on Interactive Ray Tracing,* 105–12.

Rogers, D. F., and J. A. Adams. 1990. *Mathematical Elements for Computer Graphics*. New York: McGraw–Hill.

Ross, S. M. 2002. *Introduction to Probability Models* (8th ed.). San Diego: Academic Press.

Roth, S. D. 1982. Ray casting for modeling solids. *Computer Graphics and Image Processing 18,* 109–44.

Roth, S. H., P. Diezi, and M. Gross. 2001. Ray tracing triangular Bézier patches. In *Computer Graphics Forum (Eurographics 2001 Conference Proceedings) 20*(3), pp. 422–30.

Rougeron, G., and B. Péroche. 1997. An adaptive representation of spectral data for reflectance computations. In *Eurographics Rendering Workshop 1997*. pp. 126–38.

Rougeron, G., and B. Péroche. 1998. Color fidelity in computer graphics: a survey. *Computer Graphics Forum 17*(1), 3–16.

Rousselle, F., P. Clarberg, L. Leblank, V. Ostromoukhov, and P. Poulin. 2008. Efficient product sampling using hierarchical thresholding. *The Visual Computer (Proceedings of CGI 2008) 24*(7–9), 465–74.

Rubin, S. M., and T. Whitted. 1980. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*(3), 110–16.

Ruckert, M. 2005. *Understanding MP3*. Wiesbaden, Germany: GWV-Vieweg.

Rushmeier, H. E. 1988. Realistic Image Synthesis for Scenes with Radiatively Participating Media. Ph.D. thesis, Cornell University.

Rushmeier, H. E., and K. E. Torrance. 1987. The zonal method for calculating light intensities in the presence of a participating medium. In *Computer Graphics (Proceedings of SIGGRAPH '87),* Volume 21, pp. 293–302.

Rushmeier, H. E., and G. J. Ward. 1994. Energy preserving non-linear filters. In *Proceedings of SIGGRAPH '94,* Computer Graphics Proceedings, Annual Conference Series, pp. 131–38.

Rushmeier, H., C. Patterson, and A. Veerasamy. 1993. Geometric simplification for indirect illumination calculations. In *Graphics Interface '93,* pp. 227–36.

Rusinkiewicz, S. 1998. A new change of variables for efficient BRDF representation. *Proceedings of the Eurographics Rendering Workshop,* 11–23.

Rusinkiewicz, S., and M. Levoy. 2000. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000,* Computer Graphics Proceedings, Annual Conference Series, pp. 343–52.

Saito, T., and T. Takahashi. 1990. Comprehensible rendering of 3-D shapes. In *Computer Graphics (Proceedings of SIGGRAPH '90),* Volume 24, pp. 197–206.

Samet, H. 1990. *The Design and Analysis of Spatial Data Structures*. Reading, Massachusetts: Addison-Wesley.

Sattler, M., R. Sarlette, and R. Klein. 2003. Efficient and realistic visualization of cloth. *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering,* 167–78.

Schaufler, G., and H. W. Jensen. 2000. Ray tracing point sampled geometry. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering,* pp. 319–28.

Schilling, A. 1997. Toward real-time photorealistic rendering: challenges and solutions. In *1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware,* pp. 7–16.

Schilling, A. 2001. Antialiasing of environment maps. *Computer Graphics Forum 20*(1), 5–11.

Schlick, C. 1993. A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering,* Paris, France, pp. 73–84.

Schmidt, C. M., and B. Budge. 2002. Simple nested dielectrics in ray traced images. *Journal of Graphics Tools 7*(2), 1–8.

Schneider, P. J., and D. H. Eberly. 2003. *Geometric Tools for Computer Graphics*. San Francisco: Morgan Kaufmann.

Schregle, R. 2003. Bias compensation for photon maps. *Computer Graphics Forum 22*(4). 729–42.

Schuster, A. 1905. Radiation through a foggy atmosphere. *Astrophysical Journal 21*(1), 1–22.

Schwarzschild, K. 1906. On the equlibtrium of the sun's atmosphere (Nachrichten von der Koniglichen Gesellschaft der Wissenschaften zu Gottigen). *Göttinger Nachrichten 195,* 41–53.

Segovia, B., J. C. Iehl, R. Mitanchey, and B. Péroche. 2006. Bidirectional instant radiosity. *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering,* 389–98.

Segovia, B., J. C. Iehl, and B. Péoche. 2007. Metropolis instant radiosity. *Computer Graphics Forum 26*(3), 425–34.

Séquin, C. H., and E. K. Smyrl. 1989. Parameterized ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH '89),* Volume 23, pp. 307–14.

Shade, J., S. J. Gortler, L. W. He, and R. Szeliski. 1998. Layered depth images. In *Proceedings of SIGGRAPH 98,* Computer Graphics Proceedings, Annual Conference Series, pp. 231–42.

Shevtsov, M., A. Soupikov, and A. Kapustin. 2007a. Ray–triangle intersection algorithm for modern CPU architectures. In *Proceedings of GraphiCon 2007,* pp. 33–39.

Shevtsov, M., A. Soupikov, and A. Kapustin. 2007b. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum: Proceedings of Eurographics 2007 26*(3). pp. 395–404.

Shinya, M. 1993. Spatial anti-aliasing for animation sequences with spatio-temporal filtering. In *Proceedings of SIGGRAPH '93,* Computer Graphics Proceedings, Annual Conference Series, pp. 289–96.

Shinya, M., T. Takahashi, and S. Naito. 1987. Principles and applications of pencil tracing. In *Computer Graphics (Proceedings of SIGGRAPH '87),* Volume 21, pp. 45–54.

Shirley, P. 1990. Physically Based Lighting Calculations for Computer Graphics. Ph. D. thesis, Department of Computer Science, University of Illinois, Urbana–Champaign.

Shirley, P. 1991. Discrepancy as a quality measure for sample distributions. *Eurographics '91,* pp. 183–194.

Shirley, P. 1992. Nonuniform random point sets via warping. In D. Kirk (Ed.), *Graphics Gems III,* pp. 80–83. San Diego: Academic Press.

Shirley, P., and K. Chiu. 1997. A low distortion map between disk and square. *Journal of Graphics Tools 2*(3), 45–52.

Shirley, P., and R. K. Morley. 2008. *Realistic Ray Tracing*. Natick, Massachusetts: A. K. Peters.

Shirley, P., B. Wade, P. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg. 1995. Global illumination via density estimation. In *Eurographics Rendering Workshop 1995,* pp. 219–31.

Shirley, P., C. Y. Wang, and K. Zimmerman. 1996. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics 15*(1), 1–36.

Shoemake, K. 1985. Animating rotation with quaternion curves, *Computer Graphics (SIGGRAPH '85 Proceedings),* Volume 19, pp. 245–54.

Shoemake, K. 1991. Quaternions and 4x4 matrices. In J. Arvo (Ed.), *Graphics Gems II,* pp. 351–54. San Diego: Academic Press.

Shoemake, K. 1994a. Polar matrix decomposition. In P. Heckbert (Ed.), *Graphics Gems IV,* pp. 207–21. San Diego: Academic Press.

Shoemake, K. 1994b. Euler angle conversion. In P. Heckbert (Ed.), *Graphics Gems IV,* pp. 222–29. San Diego: Academic Press.

Shoemake, K., and T. Duff. 1992. Matrix animation and polar decomposition. In *Graphics Interface '92,* pp. 258–64.

Sillion, F., and C. Puech. 1994. *Radiosity and Global Illumination*. San Francisco: Morgan Kaufmann.

Sims, K. 1991. Artificial evolution for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH '91),* Volume 25, pp. 319–28.

Sloan, P.-P. 2008. Stupid spherical harmonics (SH) tricks. *Game Developers Conference 2008*.

Sloan, P.-P., J. Kautz, and J. Snyder. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics 21*(3), 527–36.

Sloan, P.-P., J. Hall, J. Hart, and J. Snyder. 2003a. Clustered principal components for precomputed radiance transfer. *ACM Transactions on Graphics 22*(3), 382–91.

Sloan, P.-P., X. Liu, H.-Y. Shum, and J. Snyder. 2003b. Bi-scale radiance transfer. *ACM Transactions on Graphics 22*(3), 370–75.

Sloan, P.-P., B. Luna, and J. Snyder. 2005. Local, deformable precomputed radiance transfer. *ACM Transactions on Graphics 24*(3), 1216–24.

Slusallek, P. 1996. Vision—An Architecture for Physically-Based Rendering. Ph.D. thesis, University of Erlangen.

Slusallek, P., and H.-P. Seidel. 1995. Vision—an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics 1*(1), 77–96.

Slusallek, P., and H.-P. Seidel. 1996. Towards an open rendering kernel for image synthesis. In *Eurographics Rendering Workshop 1996,* pp. 51–60.

Smith, A. R. 1984. Plants, fractals and formal languages. In *Computer Graphics (Proceedings of SIGGRAPH '84),* Volume 18, pp. 1–10.

Smith, A. R. 1995. A Pixel Is Not a Little Square, a Pixel Is Not a Little Square, a Pixel is Not a Little Square! (and a Voxel Is Not a Little Cube). Microsoft Technical Memo 6, *www.alvyray.com*.

Smith, J. O. 2002. Digital audio resampling home page. *www-ccrma.stanford.edu/~jos/resample/*.

Smits, B. 1998. Efficiency issues for ray tracing. *Journal of Graphics Tools 3*(2), 1–14.

Smits, B. 1999. An RGB-to-spectrum conversion for reflectances. *Journal of Graphics Tools 4*(4), 11–22.

Smits, B., P. S. Shirley, and M. M. Stark. 2000. Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering,* pp. 307–18.

Snyder, J. 2006. *Code Generation and Factoring for Fast Evaluation of Low-order Spherical Harmonic Products and Squares*. Microsoft Research Technical Report, MSR-TR-2006-53.

Snyder, J. M., and A. H. Barr. 1987. Ray tracing complex models containing surface tessellations. *Computer Graphics (SIGGRAPH '87 Proceedings),* Volume 21, pp. 119–28.

Soupikov, A., M. Shevtsov, and A. Kapustin. 2008. Improving kd-tree quality at a reasonable construction cost. In *IEEE Symposium on Interactive Ray Tracing,* pp. 67–72.

Spanier, J., and E. M. Gelbard. 1969. *Monte Carlo Principles and Neutron Transport Problems*. Reading, Massachusetts: Addison-Wesley.

Spencer, B., and M. Jones. 2009a. Hierarchical photon mapping. *IEEE Transactions on Visualization and Computer Graphics 15*(1), 49–61.

Spencer, B., and M. W. Jones. 2009b. Into the blue: better caustics through photon relaxation. *Computer Graphics Forum (Proceedings of Eurographics 2009) 28*(2).

Stam, J. 1995. Multiple scattering as a diffusion process. In *Rendering Techniques (Proceedings of the Eurographics Rendering Workshop),* 41–50.

Stam, J. 1998. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of SIGGRAPH '98,* Computer Graphics Proceedings, Annual Conference Series, pp. 395–404.

Stam, J. 1999. Diffraction shaders. In *Proceedings of SIGGRAPH '99,* Computer Graphics Proceedings, Annual Conference Series, pp. 101–10.

Stam, J. 2001. An illumination model for a skin layer bounded by rough surfaces. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering,* pp. 39–52.

Stam, J., and C. Loop. 2003. Quad/triangle subdivision. *Computer Graphics Forum 22*(1), 79–85.

Stark, M., J. Arvo, and B. Smits. 2005. Barycentric parameterizations for isotropic BRDFs. *IEEE Transactions on Visualization and Computer Graphics 11*(2), 126–38.

Steigleder, M., and M. McCool. 2003. Generalized stratified sampling using the Hilbert curve. *Journal of Graphics Tools 8*(3), 41–47.

Steinhurst, J., G. Coombe, and A. Lastra. 2005. Reordering for cache conscious photon mapping. *Proceedings of Graphics Interface 2005,* 97–104.

Stephenson, I. 2006. Improving motion blur: shutter efficiency and temporal sampling. *Journal of Graphics Tools 12*(1), 9–15.

Stich, M., H. Friedrich, and A. Dietrich. 2009. Spatial splits in bounding volume hierarchies. *Proceedings of High Performance Graphics 2009,* pp. 7–14.

Stockmar, A. 1986. *Proposal for a data format for exchange of luminaire data (interior, exterior, and/or road lighting luminaires) under the operating systems MS-DOS 2.x/3.xx under condition of unequivocal coordination between luminaire and data set.*

Stolfi, J. 1991. *Oriented Projective Geometry*. San Diego: Academic Press.

Stroustrup, B. 1997. *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley.

Stürzlinger, W. 1998. Ray tracing triangular trimmed free-form surfaces. *IEEE Transactions on Visualization and Computer Graphics 4*(3), 202–14.

Subr, K. and J. Arvo. 2007. Steerable importance sampling. *IEEE Symposium on Interactive Ray Tracing,* pp. 133–40.

Suffern, K. 2007. *Ray Tracing from the Ground Up*. Natick, Massachussetts: A. K. Peters.

Sun, B., R. Ramamoorthi, S. Narasimhan, and S. Nayar. 2005. A practical analytic single scattering model for real time rendering. *ACM Transactions on Graphics 24*(3), 1040–49.

Sun, B., K. Sunkavalli, R. Ramamoorthi, P. Belhumeur, and S. Nayar. 2007. Time-varying BRDFs. *IEEE Transactions on Visualization and Computer Graphics 13*(3), 595–609.

Sun, Y., F. D. Fracchia, M. S. Drew, and T. W. Calvert. 2001. A spectrally based framework for realistic image synthesis. *The Visual Computer 17*(7), 429–44.

Sung, K., and P. Shirley. 1992. Ray tracing with the BSP tree. In D. Kirk (Ed.), *Graphics Gems III,* pp. 271–274. San Diego: Academic Press.

Sung, K., J. Craighead, C. Wang, S. Bakshi, A. Pearce, and A. Woo. 1998. Design and implementation of the Maya renderer. In *Pacific Graphics '98*.

Sutherland, I. E. 1963. Sketchpad—a man–machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference (AFIPS),* pp. 328–46.

Suykens, F., and Y. Willems. 2000a. Adaptive filtering for progressive Monte Carlo image rendering. In *Proceedings of WSCG 2000 Conference*.

Suykens, F., and Y. Willems. 2000b. Density control for photon maps. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering,* pp. 23–34.

Suykens, F., and Y. Willems. 2001. Path differentials and applications. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering,* pp. 257–68.

Szirmay-Kalos, L., and G. Márton. 1998. Worst-case versus average case complexity of ray-shooting. *Computing 61*(2), 103–31.

Szirmay-Kalos, L., M. Sbert, and T. Umenhoffer. 2005. Real-time multiple scattering in participating media with illumination networks. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering,* 277–82.

Tabellion, E., and A. Lamorlette. 2004 An approximate global illumination system for computer generated films. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004) 23*(3), 469–76.

Talbot, J., D. Cline, and P. Egbert. 2005. Importance resampling for global illumination. *Rendering Techniques 2005: 16th Eurographics Workshop on Rendering,* 139–46.

Tannenbaum, D. C., P. Tannenbaum, and M. J. Wozny. 1994. Polarization and birefringency considerations in rendering. In *Proceedings of SIGGRAPH '94,* Computer Graphics Proceedings, Annual Conference Series, pp. 221–22.

Teo, P., E. Simoncelli, and D. Heeger. 1997. *Efficient Linear Rendering for Interactive Lighting Design*. Department of Computer Science Technical Report, Stanford University, Report No. CS-TN-97-60.

Theußl, T., H. Hauser, and E. Gröller. 2000. Mastering windows: improving reconstruction. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization,* pp. 101–8. New York: ACM Press.

Toksvig, M. 2005. Mipmapping normal maps. *Journal of Graphics Tools 10*(3), 65–71.

Tong, X., J. Wang, S. Lin, B. Guo, and H. Y. Shum. 2005. Modeling and rendering of quasi-homogeneous materials. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1054–61.

Torrance, K. E., and E. M. Sparrow. 1967. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America 57*(9), 1105–14.

Tregenza, P. R. 1983. The Monte Carlo method in lighting calculations. *Lighting Research and Technology 15*(4), 163–70.

Trumbore, B., W. Lytle, and D. P. Greenberg. 1993. A testbed for image synthesis. In *Developing Large-Scale Graphics Software Toolkits,* SIGGRAPH '93 Course Notes, Volume 3, pp. 4-7–4-19.

Truong, D. N., F. Bodin, and A. Seznec. 1998. Improving cache behavior of dynamically allocated data structures. In *IEEE PACT,* pp. 322–29.

Tsakok, J. 2009. Faster incoherent rays: multi-BVH ray stream tracing. *Proceedings of High Performance Graphics 2009,* pp. 151–58.

Tumblin, J., and H. E. Rushmeier. 1993. Tone reproduction for realistic images. *IEEE Computer Graphics and Applications 13*(6), 42–48.

Turk, G. 1991. Generating textures for arbitrary surfaces using reaction-diffusion. In *Computer Graphics (Proceedings of SIGGRAPH '91),* Volume 25, pp. 289–98.

Turkowski, K. 1990a. *The Differential Geometry of Parametric Primitives*. Technical Note, Advanced Technology Group, Apple Computer.

Turkowski, K. 1990b. Filters for common resampling tasks. In A. S. Glassner (Ed.), *Graphics Gems I,* pp. 147–65. San Diego: Academic Press.

Turkowski, K. 1990c. Properties of surface-normal transformations. In A. S. Glassner (Ed.), *Graphics Gems I,* pp. 539–47. San Diego: Academic Press.

Turkowski, K. 1993. The differential geometry of texture-mapping and shading. Technical Note, Advanced Technology Group, Apple Computer.

Unser, M. 2000. Sampling—50 years after Shannon. *Proceedings of the IEEE 88*(4), 569–87.

Upstill, S. 1989. *The RenderMan Companion*. Reading, Massachusetts: Addison-Wesley.

van de Hulst, H. C. 1980. *Multiple Light Scattering*. New York: Academic Press.

van de Hulst, H. C. 1981. *Light Scattering by Small Particles*. New York: Dover Publications. Originally published by John Wiley & Sons, 1957.

Van Horn, B., and G. Turk. 2008. Antialiasing procedural shaders with reduction maps *IEEE Transactions on Visualization and Computer Graphics 14*(3), 539–50.

van Wijk, J. J. 1991. Spot noise-texture synthesis for data visualization. In *Computer Graphics (Proceedings of SIGGRAPH '91),* Volume 25, pp. 309–18.

Veach, E. 1996. Non-symmetric scattering in light transport algorithms. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*. Wien: Springer.

Veach, E. 1997. Robust Monte Carlo Methods for Light Transport Simulation. Ph. D. thesis, Stanford University.

Veach, E., and L. Guibas. 1994. Bidirectional estimators for light transport. In *Fifth Eurographics Workshop on Rendering,* Darmstadt, Germany, pp. 147–62.

Veach, E., and L. J. Guibas. 1995. Optimally combining sampling techniques for Monte Carlo rendering. In *Computer Graphics (SIGGRAPH '95 Proceedings),* pp. 419–28.

Veach, E., and L. J. Guibas. 1997. Metropolis light transport. In *Computer Graphics (SIGGRAPH '97 Proceedings),* pp. 65–76.

Verbeck, C. P., and D. P. Greenberg. 1984. A comprehensive light source description for computer graphics. *IEEE Computer Graphics and Applications 4*(7), 66–75.

Wald, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pp. 33–40.

Wald, I., and V. Havran. 2006. On building fast kd-trees for ray tracing and on doing that in $O(n \log n)$. In *IEEE Symposium on Interactive Ray Tracing*, pp. 61–69.

Wald, I., P. Slusallek, and C. Benthin. 2001b. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pp. 277–88.

Wald, I., T. Kollig, C. Benthin, A. Keller, and P. Slusallek. 2002. Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pp. 15–24.

Wald, I., C. Benthin, and P. Slusallek. 2003. Interactive global illumination in complex and highly occluded environments. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pp. 74–81.

Wald, I., J. Günther, and P. Slusallek. 2004. Balancing considered harmful—faster photon mapping using the voxel volume heuristic. *Computer Graphics Forum (Proceedings of Eurographics 2004) 23*(3), 595–603.

Wald, I, S. Boulos, and P. Shirley. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*(1).

Wald, I., P. Slusallek, C. Benthin, and M. Wagner. 2001a. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*(3), 153–64.

Wallis, B. 1990. Forms, vectors, and transforms. In A. S. Glassner (Ed.), *Graphics Gems I*, pp. 533–538. San Diego: Academic Press.

Walter, B., A. Arbree, K. Bala, D. Greenberg. 2006. Multidimensional lightcuts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) 25*(3), 1081–88.

Walter, B., S. Fernandez, A. Arbree, K. Bala, M. Donikian, D. Greenberg. 2005. Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 1098–1107.

Walter, B., P. M. Hubbard, P. Shirley, and D. F. Greenberg. 1997. Global illumination using local linear density estimation. *ACM Transactions on Graphics 16*(3), 217–59.

Walter, B., S. N. Pattanaik, and D. P. Greenberg. 2002. Using perceptual texture masking for efficient image synthesis. *Computer Graphics Forum 21*(3), 393–99.

Walter, B., S. Marschner, H. Li, and K. Torrance. 2007. Microfacet models for refraction through rough surfaces. In *Rendering Techniques 2007 (Proc. Eurographics Symposium on Rendering)*, 195–206.

Walter, B., K. Bala, M. Kilkarni, and K. Pingali. 2008. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing*, pp. 81–86.

Wandell, B. 1995. *Foundations of Vision*. Sinauer Associates.

Wang, J., S. Zhao, X. Tong, J. Snyder, and B. Guo. 2008a. Modeling anisotropic surface reflectance with example-based microfacet synthesis. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27*(3), 41:1–41:9.

Wang, J., S. Zhao, X. Tong, S. Lin, Z. Lin, Y. Dong, B. Guo, and H. Y. Shum. 2008b. Modeling and rendering of heterogeneous translucent materials using the diffusion equation. *ACM Transactions on Graphics 27*(1). , 9:1–9:18.

Wang, R., and O. Åkerlund. 2009. Bidirectional importance sampling for unstructured illumination. *Computer Graphics Forum (Proceedings of Eurographics 2009) 28*(2). 269–78.

Wang, R., J. Tran, and D. Luebke. 2004. All-frequency relighting of non-diffuse objects using separable BRDF approximation. In *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering),* 345–54.

Wang, R., J. Tran, and D. Luebke. 2005. All-frequency interactive relighting of translucent objects with single and multiple scattering. *ACM Transactions on Graphics 24*(3), 1202–07.

Wang, R., R. Ng, D. Luebke, and G. Humphreys. 2006a. Efficient wavelet rotation for environment map rendering. In *Eurographics Symposium on Rendering*. 173–82.

Wang, R., J. Tran, and D. Luebke. 2006b. All-frequency relighting of glossy objects. *ACM Transactions on Graphics 25*(2). 293–318.

Wang, X. C., J. Maillot, E. L. Fiume, V. Ng-Thow-Hing, A. Woo, and S. Bakshi. 2000. Feature-based displacement mapping. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering,* pp. 257–68.

Ward, G. 1991a. Adaptive shadow testing for ray tracing. In *Second Eurographics Workshop on Rendering*.

Ward, G. 1991b. Real pixels. In J. Arvo (Ed.), *Graphics Gems II,* pp. 80–83. San Diego: Academic Press.

Ward, G. J. 1992. Measuring and modeling anisotropic reflection. *Computer Graphics (SIGGRAPH '92 Proceedings),* Volume 26, pp. 265–72.

Ward, G. J. 1994a. A contrast-based scalefactor for luminance display. In P. Heckbert (Ed.), *Graphics Gems IV,* pp. 415–421. Boston: Academic Press.

Ward, G. J. 1994b. The Radiance lighting simulation and rendering system. *Proceedings of SIGGRAPH '94,* pp. 459–72.

Ward, G. and E. Eydelberg-Vileshin. 2002. Picture perfect RGB rendering using spectral prefiltering and sharp color primaries. In *The Proceedings of 13th Eurographics Workshop on Rendering,* Pisa, Italy, pp. 117–24.

Ward, G. J., and P. Heckbert. 1992. Irradiance gradients. In *Third Eurographics Workshop on Rendering,* Bristol, United Kingdom, pp. 85–98.

Ward, G. J., F. M. Rubinstein, and R. D. Clear. 1988. A ray tracing solution for diffuse interreflection. *Computer Graphics (SIGGRAPH '88 Proceedings),* Volume 22, pp. 85–92.

Warn, D. R. 1983. Lighting controls for synthetic images. In *Computer Graphics (Proceedings of SIGGRAPH 83),* Volume 17, pp. 13–21.

Warren, H. 2006. *Hacker's Delight*. Reading, Massachusetts: Addison-Wesley.

Warren, J. 2002. *Subdivision Methods for Geometric Design: A Constructive Approach*. San Francisco: Morgan Kaufmann.

Watt, A., and M. Watt. 1992. *Advanced Animation and Rendering Techniques*. New York: Addison-Wesley.

Weber, M., M. Milch, K. Myszkowski, K. Dmitriev, P. Rokita, and H.-P. Seidel. 2004. Spatio-temporal photon density estimation using bilateral filtering. *Proceedings of Computer Graphics International, 2004,* 120–27.

Weghorst, H., G. Hooper, and D. P. Greenberg. 1984. Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*(1), 52–69.

Wei, L.-Y. 2008. Parallel Poisson disk sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27*(3), 20:1–20:10.

Wei, L.-Y., S. Lefebvre, V. Kwatra, and G. Turk. 2009. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report*.

Weistroffer, R. P., K. Walcott, G. Humphreys, and J. Lawrence. 2007. Efficient basis decomposition for scattered reflectance data. *Eurographics Symposium on Rendering,* 207–18.

Westin, S., J. Arvo, and K. Torrance. 1992. Predicting reflectance functions from complex surfaces. *Computer Graphics 26*(2), 255–64.

Weyrich, T., P. Peers, W. Matusik, and S. Rusinkiewicz. 2009. Fabricating microgeometry for custom surface reflectance *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 28*(3), 32:1–32:6.

Whitted, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*(6), 343–49.

Wilkie, A., A. Weidlich, C. Larboulette, and W. Purgathofer. 2006. A reflectance model for diffuse fluorescent surfaces. In *Proceedings of GRAPHITE,* 321–31.

Williams, L. 1978. Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH '78),* Volume 12, pp. 270–74.

Williams, L. 1983. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings),* Volume 17, pp. 1–11.

Williams, A., S. Barrus, R. K. Morley, and P. Shirley. 2005. An efficient and robust ray–box intersection algorithm. *Journal of Graphics, GPU, and Game Tools 10*(4). pp. 49–54.

Wilson, P. R., M. S. Johnstone, M. Neely, and D. Boles. 1995. Dynamic storage allocation: a survey and critical review. In *Proceedings International Workshop on Memory Management,* Kinross, Scotland.

Witkin, A., and M. Kass. 1991. Reaction-diffusion textures. In *Computer Graphics (Proceedings of SIGGRAPH '91),* Volume 25, pp. 299–308.

Wolff, L. B., and D. J. Kurlander. 1990. Ray tracing with polarization parameters. *IEEE Computer Graphics and Applications 10*(6), 44–55.

Wong, T.-T., W.-S. Luk, and P.-A. Heng. 1997. Sampling with Hammersley and Halton points. *Journal of Graphics Tools 2*(2), 9–24.

Woo, A., and J. Amanatides. 1990. Voxel occlusion testing: a shadow determination accelerator for ray tracing. In *Graphics Interface '90,* pp. 213–20.

Woo, M., J. Neider, T. Davis, D. Shreiner, and the OpenGL Architecture Review Board. 1999. *The OpenGL Programming Guide*. Reading, Massachusetts: Addison-Wesley.

Woop, S., G. Marmitt, and P. Slusallek. 2006. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware 2006: Eurographics Symposium Proceedings,* Vienna, Austria, pp. 67–76.

Worley, S. P. 1996. A cellular texture basis function. In *Proceedings of SIGGRAPH '96,* Computer Graphics Proceedings, Annual Conference Series, New Orleans, Louisiana, pp. 291–94.

Wyman, D., M. Patterson, and B. Wilson. 1989. Similarity relations for anisotropic scattering in Monte Carlo simulations of deeply penetrating neutral particles. *Journal of Computational Physics 81.* 137–50.

Wyvill, B., and G. Wyvill. 1989. Field functions for implicit surfaces. *The Visual Computer 5*(1/2), 75–82.

Xu, R., and S. Pattanaik. 2005. Non-iterative, robust Monte Carlo noise reduction. *IEEE Computer Graphics and Applications 25*(2), 31–35.

Yanovitskij, E. G. 1997. *Light Scattering in Inhomogeneous Atmospheres*. Berlin: Springer-Verlag.

Yellot, J. I. 1983. Spectral consequences of photoreceptor sampling in the Rhesus retina. *Science 221,* 382–85.

Yoon, S.-E. and P. Lindstrom. 2006. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics, 12*(5), 1213–20.

Yoon, S.-E. and D. Manocha. 2006. Cache-efficient layouts of bounding volume hierarchies. In *Computer Graphics Forum: Proceedings of Eurographics 2006 25*(3), 507–16.

Yoon, S.-E., P. Lindstrom, V. Pascucci, and D. Manocha. 2005. Cache-oblivious mesh layouts. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005) 24*(3), 886–93.

Yoon, S.-E., C. Lauterbach, and D. Manocha. 2006. R-LODs: fast LOD-based ray tracing of massive models. *The Visual Computer 22*(9–11), 772–84.

Zachmann, G. 2002. Minimal hierarchical collision detection. *Proceedings of the ACM Symposium on Virtual Reality Software and Technology,* pp. 121–28.

Zeller, A. and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering, 28*(2), 183–200.

Zhou, K., Y. Hu, S. Lin, B. Guo, and H.-Y. Shum. 2006. Precomputed shadow fields for dynamic scenes. *ACM Transactions on Graphics 24*(3), 1196–1201.

Zhou, K., Q. Hou, R. Wang, and B. Guo. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 27*(5), 126:1–126:11.

Zhukov, S., A. Iones, and G. Kronin. 1998. An ambient light illumination model. In *Eurographics Rendering Workshop 1998,* pp. 45–55.

Zickler, T., S. Enrique, R. Ramamoorthi, and P. Belhumeur. 2005. Reflectance sharing: image-based rendering from a sparse set of images. *Rendering Techniques 2005 (Proceedings of Eurographics Symposium on Rendering),* 253–65.

Zimmerman, K. 1995. Direct lighting models for ray tracing with cylindrical lamps. In *Graphics Gems V,* pp. 285–89. San Diego: Academic Press.

Zinke, A., C. Yuksel, A. Weber, and J. Keyser. 2008. Dual scattering approximation for fast multiple scattering in hair. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) 27*(3), 32:1–32:10.

Zorin, D., P. Schröder, T. DeRose, L. Kobbelt, A. Levin, and W. Sweldens. 2000. *Subdivision for Modeling and Animation*. SIGGRAPH 2000 Course Notes.

Zuniga, M. and J. Uhlmann. 2006. Ray queries with wide object isolation and the S-tree. *Journal of Graphics, GPU, and Game Tools 11*(3), 27–45.

Index

Numbers

(0,2)-sequences, 368–372
 2D sample generation, 370
 defined, 368
 elementary intervals, 370
 illustrated, 369
 low-discrepancy sampler, 372
 random scrambling, 370, 371
 sample generation, 371
 sequence length, 370
 stratification, 368
 unscrambled, 370
 use of, 378
 See also low-discrepancy sampling
1D sample patterns, 344
2 × 2 linear systems, 1020
2D sample patterns, 343, 347
2D (u, v) mapping, 514–515
3D grids, 592–594
 defined, 592
 implementations, 593
 local density computation, 594
 voxel coordinates/offsets computation, 594
3D mapping, 518–519
3D sample patterns, 343
3D viewing problem, 305
4 × 4 matrices, 1020–1021

A

absorption, 575, 576–578
 cross section, 576
 defined, 575
 effect, 577–578
 illustrated, 577
 radiance and, 577–578
 total fraction of light, 578
absorption coefficient
 defined, 435
 of gold, 436
abstract base classes, 15, 41
abstraction
 efficiency versus, 991
 radiative transfer, 281
 of shapes, 107
acceleration structures
 bounding volume hierarchies (BVHs), 208–227
 defined, 183
 goal, 192
 grid accelerators, 195–208
 implementation, 172
 KD-tree accelerator, 227–245
 octrees for, 1022
 primitive subdivision-based, 250
 in ray-object intersection, 8
 specialized, 992
acceptance probability, 653
across tangents, 172
adaptive sampling, 334, 341, 385–389
 challenge, 386
 criteria example, 390
 defined, 334
 implementation, 385, 386
 tests, 388–389
 See also sampling
affine space, 55
affine transformations, 97
aggregates, 191–195
 bugs, 245–246
 bugs, finding, 246–249
 bugs, fixing, 249–250
 debugging, 245–250
 implementation testing, 246–247
 intersections, 192–193
 performance bugs, 250
 ray-box intersections, 193–195
 volume, 596–598
aliases, 331
aliasing, 331, 331–332
 checkerboard textures, 548
 defined, 324
 illustrated, 332
 from point sampling, 333
 shading, 336
 sources, 336–337
 texture, 501, 502
 visual impact, 501
alpha masks, 144
ambient occlusion (AO), 926, 927
 common use, 927
 defined, 926
 illustrated, 926
 See also precomputed light transport
angle measure conversion, 1001
angle rotation, 81, 83
animated shapes, 1065, 1066, 1067, 1068
animation
 keyframe matrix, 90
 primitives, 189–191
 transformation, 90–100
anisotropic microfacet model, 457–460
 defined, 457
 distribution function, 458
 exponents, 459
 sampling, 699–701
 spheres rendered from, 459
 See also microfacet models
anisotropic phase functions, 584
anisotropic surfaces, 425
antialiasing
 adaptive sampling, 334
 checkerboard textures, 548
 fractional Brownian motion, 561
 mix textures and, 521
 nonuniform sampling, 334
 polka dots, 559
 prefiltering, 334–335
 procedural texturing, 545, 551
 ray differentials and, 46, 68–69
 scale texture and, 521
 techniques, 332–335
 texture, 503–513
application programming interfaces (APIs)
 cleanup, 1053, 1056
 function definitions, 1054, 1055, 1057, 1058, 1060, 1063, 1065, 1069, 1070, 1072
 initialization, 1053, 1056, 1060
 local classes, 1054, 1056, 1059, 1060
 macros, 1053, 1055, 1063
 rendering option functions, 1057
 routine subsets, 1052

application programming interfaces (APIs) *(continued)*
shape recreation, 1073
states, 1052
static data, 1052, 1054, 1055, 1056, 1060
area
differential, 291, 292, 293
integrals over, 292–293
area lights, 623–627
creation, 1065
in current graphics state, 1067
defined, 623–625
diffuse, 718
emitted power, 626
illumination illustration, 624
incident illumination, 627
infinite, 627–631, 724–730, 943–949
interface, 625
projection, 943–949
protected data, 626
sampling, 358, 377–378, 715–722
sampling methods, 717–719
See also light sources; light(s)
arena-based allocation, 1014–1017
advantages, 1014
defined, 1014
memory areas, 1014–1017
See also memory allocation; memory management
arithmetic logic units (ALUs), 993
asymmetry parameter, 585, 586
atomic operations, 1036–1038
add of floating-point values, 1038
compare and swap, 1037
implementation of, 1036
lock-free algorithm, 1038
atomics, 38–39
defined, 38
mutexes versus, 39
attenuation
coefficient, 579, 874
geometric, 453, 455
light, 13
attribute stacks, 1059
axis-aligned bounding boxes (AABBs)
defined, 70
homogeneous volumes, 590
illustrated, 71
world space, 587

B

back-scattering, 585
balance heuristic, 691, 693
band-limited functions, 329, 330, 509
barycentric coordinates
computation, 141, 142
computation for points, 147
defined, 140
intersection point, 140
less than zero/greater than one, 142
base-2 logarithms, 1001–1002
basis functions, 927–932
advantages, 928
dual of, 930
efficiency, 928
integral of, 931
orthonormal, 928, 930–932
piecewise-constant, 928–930
projection onto, 930
SPD representation, 261
for spectral representation, 263
spherical harmonics (SH), 932
theory, 927–932
basis vectors, 55–56, 82
beam transmittance, 580–581
defined, 580
illustrated, 581
multiplicative property, 581
properties, 580–581
best-candidate algorithm, 379
best-candidate sampling pattern, 378–385
defined, 379
integrator sample computation, 385
sample generation, 385
stratified sampling pattern comparison, 382
table, 380
tiled sample patterns, 384
toroidal topology, 379
bias
compensation for bounding geometry, 781, 783
compensation, trace ray for, 783
introduction of, 784
photon mapping, 828
start-up, 655–656
in variance reduction, 686–687
bidirectional path tracing, 770–771
defined, 762, 770
as generalization of standard path-tracing algorithm, 770
IGI correspondence, 775
path instances construction, 851–852
radiance along path computation, 846–847
refinements, 771
See also path tracing
bidirectional reflectance distribution functions (BRDFs), 294–296
adapter, 430–431
defined, 10
delta distribution, 439–440
energy conservation, 295
Fresnel incidence effects, 460
illustrated, 295
implementation, 428
interface, 428–432
Lambertian model, 446
low-level, 477
measured, 462–470
memory management, 482–483
microfacet-based models, 447
parameterized models, 462
path, 757
public data, 429
reciprocity, 295
scaling adapter, 431–432
specular reflection, 439
utility functions, 435
bidirectional scattering distribution functions (BSDFs), 477–483
BRDF reuse adapter, 430–431
class representation, 477
complex material property surfaces, 43
defined, 11, 296
destructor, 482
for final gathering, 824
implementations, 428, 479
inline functions, 426
interface, 428–432
interface setting, 426
matrix, 977–979
memory allocation, 187
pointer storage, 705
random variables, 705
sample initialization, 705
sample offsets, 706
sample values, 705
sampled directions, 707–708
sampling, 705–708
sampling distribution, 979
sampling PDF computation, 707
scaling adapter, 431–432
special handling, 429
transfer matrix, 978
utility functions, 435
values, 46
bidirectional scattering-surface reflectance distribution function (BSSRDF), 296–297, 598–600
BRDF reflectance correspondence, 912

computation, 885
defined, 296
diffuse reflectance conversion to, 908
evaluation, 885, 908
illustrated, 297
lower-level scattering properties, 598
properties, 598
scattering equation, 296
special handling, 429
subsurface scattering materials, 599–600
in volume scattering, 598–600
bidirectional transmittance distribution functions (BTDFs)
defined, 295
denotation, 295
low-level, 477
specular transmission, 442, 443
bilinear interpolation texture, 522–523
defined, 522
See also texture(s)
binary space partitioning (BSP) trees
defined, 227
KD-trees, 227–245
octrees, 227
Blinn microfacet distribution, 455–457
defined, 455
exponent variation effect, 458
functions, 696
implementations, 695
interface, 695
Killeroo, 456
normalized functions, 456–457
sampling, 695–699
See also microfacet models
blocked 2D arrays, 1017–1020
arrangement in memory, 1017
block size, 1017
natural layout, 1018
blurring, 398
bootstrapping samples, 853, 856, 857
boundary vertices, 156
determining, 157
limit rule, 171
odd, 167
one-ring, 164–165
subdivision and, 162
tangent computation, 172–173
valance computation, 158–159
bounding
cylinder, 126
disk, 130–131
shape interface, 109–110
sphere, 116
subdivision surfaces, 160
bounding boxes
axis-aligned (AABBs), 70, 71, 587, 590
BVH nodes, 226
grid, 202
oriented (OBBs), 70
overlapping kd-tree nodes, 238
points, 71, 72
primitive, 201
processing expense, 70
radiance probes, 959
ray intersection with, 193
sphere, 73
surface area computation, 72–73
in texture space, 542
three-dimensional, 70–74
transforming, 88
world space, 109, 139
bounding spheres, 893
bounding volume hierarchies (BVHs), 208–227
binary, 208
bounds computation, 213
characteristics, 208–209
compact, for traversal, 222–224
construction, 210–217
defined, 208
definition of, 209
efficiency, 208
illustrated example, 209
implementation, 225–226
interior nodes, 213, 224, 226
leaf nodes, 211, 213, 226
linear layout in memory, 223
nodes, 211, 212, 223, 224
nodes, bounding boxes, 226
nodes, checking ray against, 225
nodes, following ray through, 225
partition axis, 214
performance, 226
primitive storage, 210
surface area heuristic, 217–222
traversal, 224–227
box filter, 394–396
defined, 394
graph, 395
isotropic triangle, 536–539
mip map, 536
reconstruction, 395
See also filters; reconstruction
BRDF. *See* bidirectional reflectance distribution function
BSDF. *See* bidirectional scattering distribution function
BSSRDF. *See* bidirectional scattering-surface reflectance distribution function
BTDF. *See* bidirectional transmittance distribution function
bugs (aggregates), 245–250
causes, 249
challenge, 245
finding, 246–249
fixing, 249–250
performance, 250
bugs, `pbrt`, 50
bump mapping, 490–496
appearance, 490–492
application effect, 490, 492
defined, 490
displacement texture evaluation, 495
implementations, 494
offset position computation, 495
offsets selection, 495–496
in shading normals computation, 492
in Sponza atrium model, 493
support, 492
See also materials
bumpy textures, 565–566
BVHs. *See* bounding volume hierarchies

C

cache coherency, 1034
cache-friendly memory usage, 1012–1013
caching
geometry, 992
texture, 524–527
camera paths, 845
connecting, 849
radiance computation, 849
vertex, reflected light computation, 850
camera space
defined, 304
origin, 304
points in, 308, 311
world space transformation to, 304
See also coordinate spaces
cameras, 5–7
class, 301–302
coordinate spaces, 303–305
defined, 4
environment, 318–319
implementation, 303
interface, 303
models, 301–321
orthographic, 306–310
perspective, 310–313

cameras *(continued)*
physical lens system simulation, 302
pinhole, 5, 6, 316
projective, 305–318
ray computation, 303
real-world, 302, 313
simulator, 6–7
candelas, 287
candidate points distribution algorithm, 892
canonical uniform random variables, 639
Cartesian coordinates, 661
caustic lighting
computation, 817
illustrated, 818
photon map, 802, 803, 817
rendering of, 802
caustic photons, 813
checkerboard textures
aliasing, 548
antialiasing, 548
application illustration, 547
check function frequency, 547
closed-form box-filtered, 549
defined, 546
filtering, 549
sampling method comparison, 348
solid, 552–553
step function integration, 550
supersampling, 551
texture coordinates, 514, 546
See also texture(s)
child octree nodes, 899
circle of confusion, 313, 314
clamping, 561
function, 1000
Monte Carlo integration and, 782
classes
abstract base, 15, 41
geometric, 55
reentrant methods, 40–41
relationships, 27, 42
summarization, 26
surface integration, 42
synchronization, 31
closed mesh, 152
code, this book
bugs, 50
optimization, 49
pointer versus reference, 48
using/understanding, 48–50
coherent shared memory, 36
color(s)
out-of-gamut, 413
RGB, 273–279
tristimulus theory, 270
XYZ, 270–273
communication with user, 1004–1006
error reporting, 1005
float file reader, 1006
reporting progress, 1005–1006
compare and swap, 1037
concentric mapping, 666, 667
conditional density function, 662
conductors, 434, 435
cones, 133–134
implicit form, 133
parametric form, 133
partial derivatives, 133–134
See also shapes
constant texture, 520
continuous coordinates, 337, 338
continuous random variables, 639–640
continuous transformations, 74
control mesh, 148, 149, 150
consistently ordered, 154
directed edge, 154
See also subdivision mesh
convex hull property, 160, 163
convolution
computing, 327
Fourier transform, 329
operation, 327
shah function, 329, 330
in spatial domain, 329
coordinate spaces
camera space, 304, 308, 311
illustrated, 304
normalized device coordinate (NDC) space, 304
object space, 115, 138, 303
raster space, 304, 309, 384
screen space, 39, 304
world space, 56, 109, 139, 191, 199, 303
coordinate systems, 55–57
defined, 55
frame, 55, 56
handedness, 56–57, 89
left-handed, 56, 57
reflection, 425
right-handed, 56, 57
shading, 425, 427
from vectors, 63
world space, 56
cosine-weighted hemisphere sampling, 668–670
CPUs
cache coherence protocol, 1034
design, 993
packet tracing on, 995
performance improvement, 993
SIMD width, 995
Cramer's rule, 140
Cranley-Patterson rotations, 384
critical angle, 439
cross product, 61–62
defined, 61
in left-handed coordinate system, 61
of perpendicular unit vectors, 62
"up" vector, 85
cross-referencing, 3–4
cube maps, 945–946
cumulative distribution function (CDF)
constructing, 643–644
defined, 639
discrete, 644
integral, 644
inversion, 644, 648
monotonically increasing, 648
normalization, 649
PDF and, 640
piecewise-constant 1D functions, 647–650
segments computation, 649
value storage, 648
current transformation matrices (CTMs), 1053–1055
active, 1053
for initialization, 1058
named copies of, 1055
`pbrt` maintenance, 1053
selective modification, 1053
use of, 1054
curse of dimensionality
defined, 350
stratified sampling and, 684–685
cylinders, 124–129
basic setting, 125
bounding, 126
coefficients computation, 127
construction, 125–126
hit point computation, 127
illustrated, 126
implementation, 125
implicit form, 126
infinitely long, 126
intersection, 126–127
parametric form, 125
partial, 127–128
partial derivatives, 128
sampling, 719
surface area, 129
See also shapes

cylindrical mapping, 517

D

dart throwing, 378
data races
 coordination and, 36–40
 defined, 36
 See also parallelization
debugging
 aggregates, 245–250
decomposition
 function, 3
 matrix, 91
 in `pbrt` image rendering, 28
 polar, 97
 screen-space, 39
 transformation, 97
delta distributions, 327, 439
 Dirac, 439, 702
 in integrand, 757–758
 light, 608
density estimation, 827–833
 defined, 828
 example illustration, 829
 kernel methods, 828
 PDF construction, 828
depth of field, 313–318
 circle of confusion and, 313, 314
 defined, 314
 focal distance, 314
 illustrated, 314, 316
 lens aperture size, 314, 315
 orthographic camera, 309
 pinhole cameras and, 316
 projective cameras, 314
design
 abstraction versus efficiency, 991
 CPU, 993
 graphics API, 1046
 retrospective, 989–993
 triangles-only alternative, 991–992
detailed balance, 653
dielectric media, 434
differential area, 291, 292, 293
differential geometry
 defined, 101
 illustrated, 101
differential irradiance, 8
 See also irradiance
differential radiance, 295
 See also radiance
diffuse area lights, 718
diffuse precomputed radiance transfer (PRT), 969–973
 diffuse reflection, 970
 reflected radiance computation, 972
diffuse reflection
 defined, 424
 illustrated, 425
 Oren-Nayar, 449–451
diffuse subsurface reflectance, 906, 908
diffusion approximation, 901–908
diffusion equation, 902–904
digital difference analyzer (DDA), 203
dipole approximation, 905
 basic setting for, 906
 boundary condition, 905
Dirac delta distribution, 439, 702
direct lighting, 741–751, 802, 848
 BSDF directions, 743
 computation, 744, 768, 853
 computation strategies, 741
 defined, 45
 estimating, 745
 implementation, 744
 importance sampling, 748
 integral, estimating, 747–751
 integrator, 741, 742, 744, 746
 light source sampling parameters, 743
 for Metropolis path vertex, 848
 in Metropolis sampling, 854, 855
 multiple importance sampling, 749, 750
 photon map, 803
 reflection from, 848
 sampling, 741, 744
 scene rendering, 742
 strategy, applying, 744
 See also indirect lighting; lighting
direct photons, 810
Direct3D, 1045
directional lights
 infinite light direction, 715
 light direction point offset, 715
 outgoing ray direction, 714
 sampling, 713–715
directional sampling, 664
directions
 differential reflected, 513
 half-angle, 469
 incident, computation, 697
 incident light, 427
 incident radiance, 294
 normalized, 43
 normals, 103
 ray, 248
 sampling, 708
 specular reflection, 441
 spherical coordinate representation, 661–662
 transformation into shading space, 479
 transmitted, 45
 transmitted ray, 445–446
 vector representation, 57
 voxel stepping, 205
discrepancy
 box, 360
 computing, 359–360
 defined, 359–361
 Halton sequence, 363
 star, 360–361
 van der Corput sequence, 362
 See also low-discrepancy sampling
discrete coordinates, 337, 338
discrete random media, 915
disks, 129–132
 basic setting, 129
 bounding, 130–131
 construction, 130
 generalized to annulus, 129–130
 hit point computation, 131–132
 illustrated, 130
 intersection, 131–132
 parameterization, inverting, 132
 parametric form, 130
 partial, 129
 partial derivatives, 132
 plane intersection computation, 131
 sampling, 665–668, 718–719
 surface area, 132–133
 See also shapes
dispersion, 432
displacement textures, 494, 495
display pixels, 323–324
distant lights, 621–623
 defined, 621
 power approximation, 622
 See also light sources; light(s)
distributions
 Blinn microfacet, 455–457, 695–699
 conditional sampling, 672
 delta, 327, 439, 608, 757–758
 exponential, 646
 fluence, 905
 geometric, 8
 half-angle, 698, 700
 image, 384
 joint, 662
 light, 8–9, 15, 902–903
 marginal sampling, 673
 particle, 805
 photon, 801
 piecewise-constant 1D, 647–650
 piecewise-constant 2D, 671–674
 Poisson, 888–896

distributions *(continued)*
power, 645–646
radiometric, 8
spectral power (SPD), 261–271, 275, 280
stationary, 653
transforming between, 660–662
dot product, 60–61
absolute value, 61
angle relationship, 61
computation between normals and vectors, 66
defined, 60
vector, 556

E

efficiency
abstraction versus, 991
bias and, 686–687
BVHs, 208
estimator definition, 679
importance sampling and, 688–693
Monte Carlo estimator, 747
Monte Carol integration, 679–737
rejection method, 650
Russian roulette and, 680–681
sample placement and, 682–686
sampling reflection functions, 693–708
splitting and, 681–682
elementary intervals, 370
elliptically weighted average (EWA), 539–544
coordinates conversion, 541
ellipse axes computation, 540
ellipse bounding box computation, 542
ellipse coefficients computation, 542
ellipse eccentricity, 540
filter illustration, 539
filter table lookup, 543
filter weights initialization, 544
filtering algorithm, 509
image pyramid, 541
lookup detail, 541
unified resampling filter, 539
emission, 575, 578, 578–579
defined, 575
differential equation, 578
illustrated, 579
emission-only integrator, 876–882
defined, 877
scene rendering, 878
stepping, 880–881
See also volume integrators
energy balance, 52, 751
energy conservation
BRDF, 295
light, 282
environment camera, 318–319
defined, 318
ray coordinate computation, 319
rendering example, 319
See also cameras
environment maps
changing, 629
for illumination, 630
See also infinite area lights
environmental projection, 318
equation of transfer, 873–875
defined, 873
diffusion approximation, 901–908
incident radiance, 875
Monte Carlo integration and, 730
radiance effects, 875
error reporting, 1005
Euler decomposition, 953–956
even vertices, 162
boundary rule application for, 165
one-ring rule for, 163
EWA. *See* elliptically weighted average
exitant radiance, 286–287
computing from photons for diffuse surface, 832
computing from photons for glossy service, 832
computing with radiance photons, 824, 826
defining, 754
See also radiance
expected values, 640, 641, 654
exponential density, 594–596
defined, 594
illustrated, 595
perpendicular projection, 596
exponential distribution, 646
extraordinary vertices
defined, 152
determination, 157
See also vertices

F

false sharing, 1035
film, 402–413
effect of, 402
image, 404–412
image output, 412–413
implementations, 402, 404
interface, 403–404
filters
box, 394–396, 536
extent in each direction, 393
functions, 393–402
Gaussian, 396–398
Mitchell, 398–400
`pbrt` implementations, 393, 394
pixel reconstruction, 393
reconstruction, 531
sinc, 392, 400–402
support, 394
triangle, 396
unified resampling, 539
weight table computation, 407
width definition, 393
See also reconstruction
final gathering, 819–827
BSDF for, 824
defined, 802
illustrated, 819
one-bounce, 819
photon mapping with/without, 820
ray direction selection, 821
rays, 826–827
fixed-function graphics hardware, 995
float file reader, 1006
floating-point
error, 112
integer conversion, 1002
weights, 32
fluence distribution, 905
fluorescence, 282
flux. *See* radiant flux
focal distance, 314
focal plane, 317–318
foreshortening
defined, 306
perspective projection, 310
forward differencing, 494
Fourier analysis, 329
concept, 325–327
defined, 324
equation, 326
pairs, 327
use of, 326
Fourier transform, 325–327
convolution, 329
defined, 325, 326
inverse, 326
one-dimensional function, 326
product of two functions, 329
shah function, 329
fractional Brownian motion, 561–562
antialiasing, 561
graphs, 562
implementation, 561

noise, 563
noise octave computation, 563
fragments
+= symbol, 3
content expansion, 3
defined, 2
initialization, 2–3
page numbers, 4
series of, 3
frames
defined, 55
standard, 56
transformation of, 74
frequency domain
defined, 325
sampled signal, 329
transforming to spatial domain, 326
Fresnel equations
for conductors, 434, 435, 436
defined, 434
for dielectric media, 434, 437
Fresnel incidence effects, 460–462
diffuse term derivation, 461
illustrated, 461
spectra, 460
Fresnel interface, 439
Fresnel reflectance, 434–439
computation for dielectric, 434
for conductors, 434, 436–437
for dielectric media, 434, 437–439
diffuse, 905
for parallel polarized light, 434
for unpolarized light, 435
See also reflectance
Fresnel transmittance, 909
functions
angle measure conversion, 1001
band-limited, 329, 330, 509
base-2 logarithm, 1001–1002
checkerboard, 550
clamping, 1000
conditional density, 662–663
decomposition, 3
expected value, 640
floating-point to integer conversion, 1002
fractional Brownian motion, 561–562
frequency, 326
frequency space representation, 326
image contribution, 836
image texture, 538
importance, 799
incident radiance, 981
linear interpolation, 1000
lookup, 1032, 1033, 1050
marginal density, 663
memory allocation, 1013
microfacet distribution, 456–457, 696
modulus, 1001
noise, 553–554
phase, 582, 583–587
piecewise-constant 1D, 647–650
ray-casting, 752
reentrant, 41
reflectance distribution, 424
rendering option, 1057
sampling reflection, 693–708
scalar contribution, 837
as series of fragments, 3
smooth interpolating, 563
spectrum, 329, 330, 331
texture, 501, 509–510
transformation, 1054
turbulence, 563–565
utility, 1000–1003
variance of, 640, 641
future, 995

G

gamma correction, 526
garbage collection, 107–108
Gauss-Jordan elimination, 1021
Gaussian filter, 396–398
defined, 396
graph, 397
utility functions, 398
See also filters; reconstruction
generalized nth nearest-neighbor estimate, 829
geometric attenuation, 453, 455
geometric classes, 55
geometric distribution, 8
geometric optics
defined, 281
linearity assumption, 295
as reflection model source, 424
geometric primitives, 187–189
allocation, 187
with animated transformation, 100
collection, 22–23
methods, 188–189
geometry
caching, 992
differential, 101–103
shading, 112–113
specular transmission, 445
surface, approximating, 506
Gibbs phenomenon, 336, 337
glass material, 490
global illumination
algorithms, 751
bidirectional path tracing, 770–771
instant, 772–784
irradiance caching, 784–797
Metropolis light transport, 833–861
path tracing, 760–771
photon mapping, 797–833
See also indirect lighting
global include files, 999, 1000
global variables, 2–3
glossy precomputed radiance transfer (PRT), 974–982
BSDF matrix, 977–979
defined, 974
implementations, 980
integrator, 979–982
rendering with, 975
steps, 974
transfer matrix, 975–977
glossy specular reflection
defined, 424
illustrated, 425
See also specular reflection
goniophotometric diagram lights, 618–621
defined, 618
example, 619
for real-world light sources, 619
sampling, 713
scene rendering, 620
gradients
contributions, 557
vectors, 556
graphics processing units (GPUs), 993
graphics state, 1059–1061
current, 1060
defined, 1059
hierarchical, 1059–1061
methods, 1067
pushed, 1071
storage, 1059
grid accelerator, 195–208
creation, 197–202
defined, 195
grid bounding box, 202
grid resolution, 198
grid size, 198
implementation, 196
initialization, 196
number of voxels, 198, 199
performance, 196
ray entrance, 202
space division, 196
“teapot in a stadium” problem, 196
traversal, 202–208
See also acceleration structures

H

half-angle vectors, 452, 697, 700
Halton sampler, 363–366
 stratified sampler comparison, 365
Halton sequences, 362–368
 characteristics, 363
 discrepancy, 363
 drawbacks, 366
 points, 364
 randomized, 366–368
 sample generation with, 365–366
Hammersley sequences, 362
 drawbacks, 366
 points, 364
hemispherical-directional reflectance, 430, 447
hemispherical-hemispherical reflectance, 430
 estimation, 704
 Lambertian BRDF, 447
Henyey-Greenstein phase function, 585–586
 asymmetry parameter, 585, 586
 defined, 585
 plot, 587
 weighted sum, 586
heterogeneous scattering, 886
hierarchical graphics state, 1059–1061
hierarchical integration, 908–912
 in refection evaluation, 908, 910
 rendering with, 908–912
histogramming, 828
homogeneous coordinates, 75–76
 defined, 75
 transformation, 75–76
 weight, 75
homogeneous volumes, 589–591
 AABBs, 590
hyperboloids, 134–135

I

ideal sampling, 327–331
identity matrix, 76, 87
identity transformation, 76
illumination algorithms, 751
image contribution function, 836
image file input/output, 1003–1004
image film, 404–413
 color alternatives, 406
 configuration, 405
 crop window, 405–406
 discrete integer pixels, 409
 extent computation, 406
 filter table offsets, 410
 filter weight table computation, 407
 looping, 409, 410
 mutex use, 410
 pixel updates, 411
 pixel values, 408
 pixel weight, 409
 storage allocation, 407
 XYZ color, 406
image pixels, 323
image pyramids
 defined, 529
 EWA filter, 541
 implementation, 530
image sampling interface, 338–346
image textures, 523–544
 caching, 524–527
 defined, 523
 elliptically weighted average (EWA), 539–544
 example illustration, 523
 function value computation, 538
 isotropic triangle filter, 536–539
 memory requirements, 524
 memory use, 534
 mip maps, 528–536
 parameterization, 524
 See also texture(s)
image(s)
 distribution, 384
 filtering, 391
 reconstruction, 389–402
 resampling, 531
 resizing, 531
 resolution, 340, 342, 531
 synthesis, 335–336
 tile shapes, 341
implicit form
 cones, 133
 cylinders, 126
 hyperboloids, 134
 paraboloids, 134
 spheres, 114
importance functions, 799
importance sampling, 688–693, 734
 defined, 679
 direct lighting, 748
 ease of use, 690
 Monte Carlo estimator convergence, 688
 multiple, 690–693
 variance and, 688
in-scattering, 581–582, 581–583
 defined, 579
 illustrated, 583
 phase function, 582
 radiance increase, 582
 source term, 582
 See also scattering
incident light, 427
incident photons, 831
incident radiance, 286–287, 288, 442, 744
 as continuous function, 323
 convolving, 968
 direction, 294
 equation of transfer, 875
 function, 981
 function projection, 949–950
 sampled direction and, 819
 SH coefficients computation, 964
incident radiance probes, 963
include files
 global, 999, 1000
 main, 999–1000
index of refraction
 for conductors, 435
 defined, 432
 of gold, 436
 variance, 432
indexing, 3–4, 153
indirect lighting, 784, 802
 component example, 785
 estimating, 789
 illumination computation, 783
 irradiance cache and, 789
 photon map, 803
 photon map integrator, 818
 rendering, 772
 sampling, 776
 See also direct lighting; global illumination; lighting
indirect photons, 810, 812, 813, 821, 823
inequality operators, 1021
infinite area lights, 627–631, 943–949
 cube maps with, 948
 defined, 627
 for environment lighting, 627
 illumination illustration, 628
 PDF initialization, 724–725
 projection, 943–949
 ray PDF computation, 730
 sample point conversion to direction, 728
 sampled direction PDF computation, 728
 sampling, 724–730
 sampling steps, 724
 total power, 631
 utility classes, 948

See also area lights; light sources; light(s)
initialization
APIs, 1053, 1056
with CTM, 1058
current graphics state, 1060
instancing, object, 189–191
defined, 189
memory use, 189
in scene description, 169–170
shapes with, 183
instant global illumination (IGI) integrator, 772–784
bidirectional path tracing and, 775
computation, 773
defined, 772
indirect lighting rendering, 772
local structures, 779
number of light sets variation with, 774
parameters, 773
path connections, 775
virtual light source creation, 775–780
integrals
basis function, 931
direct lighting, 747–751
estimating with Metropolis sampling, 656
evaluation, 759
over area, 292–293
over paths, 755–757
over points, 756
over projected solid angles, 289–290
over spherical coordinates, 290–292
of product of three or more functions, 932
radiometric, 288–293
triple product, 984
XYZ color, 280
integrands
delta distributions in, 757–758
partitioning, 758–759
integrators
defined, 739
direct lighting, 741, 742, 744, 746
emission-only, 876–882
glossy PRT, 979–982
instant global illumination (IGI), 772–784
interfaces, 740
irradiance caching, 786–788
multiple sample use, 354
path, 763, 766–767
path tracing, 760
photon, 800–803
radiance probes, 966
sample patterns and, 344
samples for, 343, 354, 376
simulating light propagation, 26
single scattering, 882–885
stratified sample generation for, 356
subsurface, 887–888
surface, 42, 739
utility functions, 709
volume, 876–885
for Whitted ray tracing, 41–47
intensity, 284–285
defined, 284
equation, 285
solid angle and, 284–285
interior BVH nodes, 213, 224, 226
interior kd-tree nodes
field usage, 229
information access, 229
initialization, 234
memory, 229
processing, 242
split access position, 236
storage, 230–231
interior octree nodes, 900, 901
interior vertices, 162
limit rule, 171
one-ring vertices for, 164
tangent computation, 172
updating positions of, 162
interpolation
barycentric, 146
keyframe matrix, 90, 91
linear, 1000
mip map level, 537
photon, 827–833
rotations, 91, 99
scale, 91
scale matrix, 99
SPDs, 265
spherical linear, 94–96
translation, 91
interreflection
microfacet models and, 448
Oren-Nayar model and, 449
intersections
aggregate, 192–193
existence of, 23
inconsistencies between, 249
information, 111
photon-surface, 808
ray-box, 193–195
ray-cylinder, 126–127
ray-disk, 131–132
ray-object, 7–8
ray-plane, 507
ray-primitive, 34, 43
ray-shape, 112
ray-triangle, 140–145
routines, reading/writing, 110–111
self, 111–112
shadow rays, 208
shape interface, 110–111
sphere, 116–119
testing for, 183
time spent finding, 990
inverse Fourier transform, 326
inverse transformations, 74
inversion method, 643–650
defined, 644
exponential distribution example, 646
generalization of, 661
piecewise-constant 1D functions, 647–650
power distribution example, 645–646
use illustration, 645
See also random variables
irradiance, 282–284
candidate samples, 792, 793
computation, 794
defined, 282
differential, 582
equations, 283, 284
estimation, 795–796
integral, 290, 292, 293
Lambert's law, 283
at points, 288, 898
sample region determination, 796–797
sample storage, 792
sample weighting, 794
samples, 791–797, 898
irradiance caching, 784–797
BSDF partitioning, 789
defined, 784
design issues, 784
implementation, 785
indirect lighting and, 789
integrator, 786–788
lookup and interpolation, 791–794
path tracing, 795
prime, 788
rendering with, 789–791
sample storage, 788
values, adding, 794–797
irregular isotropic measured BRDFs, 463–466
defined, 463
mapping function, 464
shortcomings, 464
See also measured BRDFs

isotropic phase functions, 583
isotropic surfaces, 424–425
isotropic triangle filter, 536–539

K

kd-tree accelerator, 227–245
 defined, 227
 illustrated, 228
 traversal, 240–245
 tree-building algorithm, 231
 See also acceleration structures
kd-trees, 1027–1033
 as binary trees, 229
 building of, 228
 child node initialization, 240
 child node processing, 1033
 construction, 231–240, 1029
 creation to store set of points, 1030
 defined, 227, 1027
 depth, 232, 233, 240
 depth-first traversal, 1033
 interior nodes, 229, 230
 intersection tests, 244–245
 kd-tree accelerator versus, 1028
 leaf nodes, 229, 230, 1030
 lookups, 1032
 node allocation, 233
 node array, 241
 node, bounding boxes overlapping, 238
 node children pointers, 243
 node layout, 231
 node memory, 230–231, 232, 233
 node storage, 230
 node surface area, 236
 node type decision, 232
 nodes, allocating, 1031
 parameterization, 1029
 parameters, 232–233
 photon storage, 821
 primitive indices, 232
 representation, 229–231
 split axis selection, 236
 split preference, 234
 traversal of rays through, 241
 working memory allocation, 235–236, 240
kernel methods, 828
keyframe matrices, 90, 91
Killeroo model
 Blinn microfacet distribution, 456
 Lambertian model, 450
 measured BRDFs, 463
 metal material, 491
 Oren-Nayar model, 450
 plastic material, 486
 specular reflection, 433
 specular refraction, 433
 subdivision surfaces, 148–149
 subsurface material, 491
 Torrance-Sparrow, 456
Kronecker delta function, 930

L

Lambertian reflection, 43, 446–447
 BRDF, 446
 Killeroo model, 450
 See also reflection
Lambert's law, 283
Lanczos window, 400
lat-long map projection, 944
Latin hypercube sampling (LHS)
 advantage, 355
 defined, 355
 illustrated, 355
 sample generation, 356
 sample permutation, 356
 stratified sampling versus, 356
 See also sampling
leaf BVH nodes, 211, 213, 226
leaf kd-tree nodes
 creation, 233, 1030
 dynamic memory allocation and, 230
 field usage, 229
 initialization, 230
 intersection tests, 244–245
 memory, 229
 overlapping primitives, 229
 primitive checking in, 244
 primitive id storage, 230
 storage, 230
leaf octree nodes, 899, 900, 901
left-handed coordinate system
 cross product in, 61
 defined, 56
 illustrated, 57
 rotation matrix, 80–81
 transformations changing, 89
Legendre polynomials, 932
 computation steps, 937
 implementation, 936
 recurrence, 938, 939
 values computation, 936
LHS. *See* Latin hypercube sampling
light distribution, 8–9
 after many scattering events, 903
 defined, 4
 geometric, 8
 in media with high albedos, 902
 radiometric, 8
 spotlight, 15
light sampling, 709–730
 directional lights, 713–715
 goniophotometric lights, 713
 interface, 709–711
 lights with singularities, 711–815
 methods, 710–711
 point lights, 711–712
 projection lights, 713
 spotlights, 712–713
light sources, 605–634
 area lights, 8, 623–627, 943–949
 directional lights, 713–715
 distant lights, 621–623
 general, 942–943
 goniophotometric lights, 618–621, 713
 illumination detection, 608
 implementations, 605
 incident direction, 607
 infinite area lights, 627–631
 interface, 605–608
 intersection between ray origin and, 9
 isotropic, 610
 point lights, 285, 711–712, 942
 primitives as, 188
 projection, 941–949
 projection lights, 713
 sample default number, 606
 sampling, 708–730
 scene, 23
 scene definition, 1064–1065
 spotlights, 611–614, 712–713
 surface energy, 10
 texture projection lights, 614–618
 total power emitted, 611
 virtual, 775–780
 visibility testing, 608–609
light transport equation (LTE), 751–760
 analytic solutions, 753–754
 basic derivation, 751–752
 brevity, 753
 closed form, 739
 conservation of power, 751
 defined, 11, 751
 delta distribution in integrand, 757–758
 energy balance, 751, 752
 equation of transfer and, 873
 evaluation of, 751
 integral over paths, 755–757
 measurement equation and, 759–760
 outing radiance distribution, 753
 partitioning integrand, 758–759
 path of multiple rays, 67
 path tracing, 760–771
 surface form, 754–755, 754–757

unbiased, 784
volume, 13
lighting
caustic, 802, 803, 817
direct, 45, 741–751, 802
indirect, 772, 776, 802
volumetric. *See* radiance probes
light(s)
area, 623–627, 715–722
arrival direction, 607
attenuation, 13
behavior at wavelengths, 282
cosine falloff, 8
delta distribution, 608
differential flux, 284
directional, 713–715
directional distribution, 285
distant, 621–623
energy, 8
energy conservation, 282
fluorescence and, 282
goniophotometric, 618–621, 713
incident, 427
infinite area, 627–631
leaks, 480
linearity behavior, 282
multiple, 9
paths, 850, 851
phosphorescence and, 282
point, 609–621, 711–712, 942
polarization and, 282
projection, 713
propagation, simulating, 25
in ray radiance, 35
with singularities, 711–715
spotlights, 15, 611–614, 712–713
steady state behavior, 282
subsurface transport, 187
texture projection, 614–618
total emitted power, 608
linear interpolation, 1000
linear transformations, 74
linearity
geometric optics, 295
light behavior, 282
of light transport, 926
literate programming, 1–4
cross-referencing, 3–4
defined, 1
as enhanced macro substitution package, 3
features, 2
indexing, 3–4
influences, 2
local exploration, 833
local illumination algorithms, 751
local support, 163
lock-free algorithm, 1038
look-at transformation, 84–85
defined, 84
finding entries of, 84
See also transformations
lookup functions, 1032, 1033, 1050
Loop subdivision surfaces, 150
convex hull property, 160
implementation, 152
local support, 163
manifold control mesh, 152
modified rules, 160
refinement process, 150
rules, 150
scheme, 152
vertices, 150
weights, 150
low-discrepancy sampler, 372–378
(0,2) sequence, 372
for area lighting sampling, 377–378
defined, 372
drawbacks, 378
sample copies, 376
sample generation, 375
sample value computation, 375–376
scrambled van der Corput sequences, 373
stratified sampler comparison, 373
low-discrepancy sampling, 359–378
(0,2)-sequences, 368–372
Halton sequence, 362–368
Hammersley sequence, 363
radical inverse, 361–363
Sobol' sequence, 372
underlying uniformity, 361
van der Corput sequence, 362, 368, 372
See also sampling
luminance, 287–288
defined, 272, 287
representative values, 288
spectral response curve and, 287
units, 287

M

main rendering loop, 26–35
beginning of, 27–28
class relationships, 26, 27
defined, 26
tasks, number of, 28–30
See also `pbrt` rendering system
Malley's method, 669
marble, 567–569
defined, 567
illustrated, 568
parameters, 568
See also noise; texture(s)
masking
microfacet models and, 448
Oren-Nayar model and, 449
materials, 477–499
bump mapping, 490–496
default, 1064
description, 1063–1064
displacement function association, 492
glass, 490
interface/implementations, 483–490
matt, 484–486
measured, 489
metal, 490, 491
mirror, 490
mix, 487–488
parameters, 1061–1062, 1064
plastic, 486–487
shape, 1067
substrate, 490
subsurface, 490, 491
translucent, 483, 490
uber, 490
mathematical routines, 1020–1021
2×2 linear systems, 1020
4×4 matrices, 1020–1021
matrix decomposition, 91
matte material, 484–486
defined, 484
texture evaluation, 486
See also materials
Maxwell's equations, 281, 584
mean free paths, 807
measured BRDFs, 462–470
forms, 463
irregular isotropic, 463–466
Killeroo model, 463
regular half angle, 467–470
rendering with, 462
See also bidirectional reflectance distribution functions (BRDFs)
measured material, 489–490
measurement equation, 759–760
defined, 759
in pixel measurements, 760
memory
atomic operations, 38–39
barrier instructions, 1034
blocked layout, 1017
BSDF, management, 482–483
cache-friendly usage, 1012–1014
coherence models, 1034–1035
coherent shared, 36

memory *(continued)*
image maps, 524, 534
kd-tree nodes, 229, 230–231, 232, 233
linear layout of BVH in, 223
object instancing and, 189
overhead, reducing, 992
memory allocation
arena-based, 1014–1017
BSDF, 187
custom routines, 201
functions, 1013
preallocation, 340
samples, 345–456
variable stack, 1009
memory arenas, 1014–1017
defined, 1014
function, 1015
getting block of memory for, 1016
vectors, 1015
memory management, 1009–1020
arena-based allocation, 1014–1017
blocked 2D arrays, 1017–1020
cache-friendly memory usage, 1012–1014
complexity, 1009
reference-counted objects, 1010–1012
variable stack allocation, 1009
metal material, 490, 491
metamers, 270
Metropolis Light Transport (MLT), 833–861
advantages, 833
bootstrapping, 856
defined, 833
implementation, 857, 859
light-carrying paths, 833
local exploration, 833
mutations, 839–842
path contributions, 845–852
path generation, 842–844
path tracing comparison, 834, 835
path vertex, 848
rendering, 857–861
sample generation, 853
sample representation, 838–839
sample structure, 838, 839
sample variables, 858
in sample vector generation, 836
Metropolis renderer, 852–857
responsibilities, 853
Metropolis sampling, 652–659
acceptance probability, 653
algorithm pseudocode, 653–654
basic algorithm, 652–654
defined, 652
detailed balance property, 653
direct lighting in, 854, 855
disadvantages, 652
expected values, 654
graph illustration, 657
integrals estimation, 656
mutation strategies, 654–655, 657
one-dimensional setting example, 656–659
PDFs and, 655
sample generation, 658
start-up bias, 655–656
starting, 853
stationary distribution, 653
strategies comparison, 659
See also sampling
microfacet models, 447–460
anisotropic, 457–460
Blinn, 455–457
components, 448
defined, 447
function, 448
geometric effects, 448
interreflection and, 448
masking and, 448
Oren-Nayar, 449–451
reflection computation, 448
shadowing and, 448
Torrance-Sparrow, 43, 452–455
See also reflection; reflection models
microfacets
defined, 447
differential area of, 452
half-angle vector, 452
surface comprised of, 447
Mie scattering, 584
mip maps, 528–536
box filter, 536
defined, 528–529
filter weights, 533
image pyramid, 529, 530
image resampling weights computation, 533
level computation, 537
level initialization, 534, 536
level selection, 537
level trilinear interpolation, 537
operation support, 529
parameterization, 529
projection light, 616
reconstruction filter, 531
texels, 532, 534, 535
See also image textures
mirror material, 490
Mitchell filter, 398–400
defined, 398
graph, 397
illustrated, 399
negative lobes, 398
one-dimensional filter function, 398–399
ringing/blurring trade-off, 398
See also filters; reconstruction
mix material, 487–489
defined, 487
See also materials
mix textures, 521–522
defined, 521
evaluation, 521
See also texture(s)
MLT. *See* Metropolis Light Transport
Möbius strip, 154
modulus function, 1001
Monte Carlo algorithms, 46, 608
improvements to, 640
results averaging, 637
Monte Carlo estimator, 641–643
convergence, 643
convergence rate, 643
defined, 641
efficiency, 747
error, 643
evaluation, 643
expected value, 641
extending to multiple dimensions, 642
multiple import sampling, 691
for scattering equation, 702
Monte Carlo integration, 24–25, 458, 637–737
background, 638–641
clamping and, 782
concepts, 637–676
defined, 638
disadvantage, 638
efficiency improvement, 679–737
equation of transfer and, 730
integral estimation, 789, 977
random samples for, 745
randomness, 637
SH coefficient value estimation, 942
Monte Carlo sampling, 441, 630
artifacts, 638
random samples for, 481
multidimensional transformations, 662–674
conditional density function, 662–663
cosine-weighted hemisphere sampling example, 668–670
piecewise-constant 2D distributions example, 671–674
sphere sampling example, 663–665
triangle sampling example, 670–671
unit disk sampling example, 665–668

See also transformations
multiple importance sampling (MIS), 690–693
balance heuristic, 691, 693
defined, 691
direct lighting, 749, 750
Monte Carlo estimator, 691
power heuristic, 692, 693
sample weighting, 691
variance reduction, 691
weight computation, 825–826, 827
See also importance sampling
multipoles, 915
mutation strategies, 654–655, 657
mutexes, 1038–1039
instances, 1038
locks, 1039
reader-writer, 1039
mutual exclusion, 37, 38

N

n-rooks sampling. *See* Latin hypercube sampling (LHS)
nearest-neighbor techniques, 829
noise, 553–569
as band-limited function, 560
for fractional Brownian motion, 563
frequency content, 560
functions, 553–554
idioms, 560–565
marble, 567–569
octave of, 561
Perlin, 554–558
random polka dots, 558–560
turbulence, 563–565
value, 554
windy waves, 566–567
non-a-number (NaN) values, 57–58
nonuniform sampling, 334, 391
nonuniform scaling, 79
normalization
CDF, 649
factor, 932, 941
vector, 62–63
normalized device coordinate (NDC) space, 304, 319
normalized directions, 43
normals, 65–66
for closed shapes, 103
defined, 65
dot product computation between, 66
implementation, 65
microfacet, 448
orientation, reversing, 103
partial derivative of, 121–122
shading, 146, 480, 481
transforming, 86–88
noweb system, 2
Nyquist frequency, 332, 389
Nyquist limit, 334, 562

O

object instancing, 189–191
defined, 189
memory use, 189
in scene description, 1069–1070
shapes with, 183
object space
defined, 303
spheres in, 115
triangle mesh, 138
See also coordinate spaces
object subdivision, 192
object(s)
default material, 1064
foreshortening, 306, 310
implementations, adding, 1073
instances, 1068
layout in memory, 1013
reference-counted, 1010–1012
scene, 1072
octave, noise, 561
octrees, 1022–1027
for acceleration structures, 1022
applications, 1022
callback, 1027
child node numbering scheme, 1026, 1027
child nodes, 899, 1026
clustered irradiance samples, 898
declaration, 1022–1023
defined, 227, 1022
interior nodes, 900, 901
of irradiance points, 910
leaf nodes, 899, 900, 901
quadtree, 1025
sample point, 896–901
odd vertices, 162
offset rays, 505
one-ring
for boundary vertex, 164–165
defined, 162
for even vertex, 163
for interior vertex, 164
rule, 162
one-to-one transformations, 74
open mesh, 152
OpenEXR standard, 1003–1004
OpenGL, 1045
optical thickness, 581
computing, 731–733
defined, 731
estimation, 732
Oren-Nayar diffuse reflection, 449–451
cosine term computation, 451
defined, 449
Killeroo model, 450
shadowing, masking, interreflection and, 449
sine and tangent term computation, 451
See also microfacet models; reflection
organization, this book, 47–48
oriented bounding boxes (OBBs), 70
orthogonal matrix, 80
orthogonality, 930
orthographic camera, 306–310
church model image, 308
defined, 306
depth of field, 309
differential rays computation, 307
illustrated, 307
ray creation, 309
viewing transformation, 307
See also cameras; projective camera models
orthographic projection, 307
orthonormal basis functions, 930–932
properties, 928, 930
space of functions representation, 931
out-of-gamut colors, 413
out-scattering, 579–581
beam transmittance, 580–581
defined, 579
illustrated, 580
probability, 579
radiance and, 580
See also scattering

P

packet tracing, 995
paraboloids, 134
parallelism, 1033–1041
atomic operations, 1036–1038
expression style, 1040
memory coherence models and performance, 1034–1135
mutexes, 1038–1039
task system, 1040–1041
parallelization, 35–41
challenge, 35–36
conventions, 39–40
data races and coordination, 36–39
reentrancy expectations, 40–41

parallelization *(continued)*
See also pbrt rendering system
parallelograms, 62
parameter sets, 1047–1051
adding to, 1048–1049
defined, 1047
item structure, 1048
methods, 1050
parameters, 1047
values, looking up, 1049–1051
parametric form
cones, 133
cylinders, 125
disks, 130
hyperboloids, 134–135
paraboloids, 134
rays, 7, 67
spheres, 114
triangles, 142, 143–144
partial cylinders, 127–128
partial derivatives
cone, 133–134
cylinder, 128
disk, 132
estimation with offset rays, 505
hyperboloid, 135
paraboloid, 134
parametric, 120
position and normal, 102
shading normal, 146
sphere, 121–122
triangle, 142, 143
partial spheres, 119–121
participating media, 575
particle tracing, 798–800
defined, 797
illustrated, 798
samples of illumination, 798
theoretical basis, 798–800
particles
distribution, 805
intersections, 807
paths, 803
weights, 799, 805
partitioning
integrands, 758–759
irradiance cache integrator, 789
photon mapping, 801, 817
radiance photons, 816
partitioning primitives, 213–217
with approximate SAH, 219
axis selection, 213–214
bucket boundary, 221
equally-sized subsets, 217
goal, 214
interior nodes, 213
through node midpoint, 215–216
See also bounding volume hierarchies (BVHs)
path tracing, 760–771
bidirectional, 762, 770–771
defined, 760
implementation, 765–770
incremental path construction, 765
integrator, 760
irradiance cache integrator, 795
Metropolis Light Transport (MLT) comparison, 834, 835
overview, 762–763
path sampling, 763–764
process, 762
scene rendering, 761
paths
bidirectional, weight computation, 852
BSDF, 757
camera, 845, 849
connecting, 775, 849
contributions, 845–852
generating, 842–844
incremental construction, 765
indirect visibility, 961
integral over, 755–757
integrator, 763, 766–767
length, 849, 850
light, 851, 852
light-carrying, 833
mean free, 807
Metropolis, 850
particle, 803
photon, 806, 808, 811
radiance computation along, 846
random ray, 891
ray, vertex generation, 843
sampling, 763–764
termination, 769, 780, 807, 844
total number of, 804
vertex, finding, 769–770
vertex information, 844
pbrt rendering system
abstract base classes, 15
bugs, 50
code use, 48–50
conventions in, 39–40
integrator for Whitted ray tracing, 41–47
interface types, 16
main() function, 18–19
main rendering loop, 26–35
overview, 15–47
parallelization of, 35–41
parsing phase, 16–17
phases of execution, 16–17
reconstruction in, 391
reentrancy expectations in, 40–41
rendering loop, 17
scene file format, 16–17
scene representation, 18–24
source code distribution, 16
volumetric primitives support, 23
perfect specular reflection
defined, 424
illustrated, 425
performance bugs, aggregate, 250
Perlin noise, 554–558
characteristics, 555
computation, 554
data, 557
defined, 554
gradient weights computation, 556
illustrated, 555
implementation, 555
nested permutations, 557
weights interpolation computation, 558
See also noise
perspective camera, 310–313
implementation, 310
rendering example, 308
See also cameras; projective camera models
perspective projection
distances/angles and, 310
foreshortening, 310
offset ray computation, 313
parallel lines and, 310
perspective viewing, 310
ray origination, 312
transformation matrix, 311
transformation steps, 311–312
phase functions, 582, 583–587
anisotropic, 584
defined, 582
Henyey-Greenstein, 585–586
implementations, 584
isotropic, 583
probability density for sampling, 731
reciprocal property, 583
sampling, 731
Schlick, 586–587
weighted sum, 586
writing convention, 584
phenomenological reflection models, 423
phosphorescence, 282
photometry, 287
photon integrator, 800–803
caustic photon map, 817

defined, 800
illumination strategies, 802
indirect lighting computation, 818
initial photon subset selection, 813
local definitions, 822
merging photon data in, 812
parameters, 802–803
shared photon map data structures, 811
photon mapping
bias, 828
defined, 797
final gathering, 802, 819–827
partitioning, 801, 817
photon weights and, 805
scene rendering, 801
photon maps, 803–827
building, 803–817
caustics, 803
defined, 800
direct lighting, 803
indirect lighting, 803
lookup, 831
populating, 809
precomputed radiance values, 813–817
shared data structure updates, 811–813
types of, 803
update determination, 809
using, 817–827
photon-surface intersections, 808
photons
caustic, 813
close to lookup point, 821
density, 801
depositing at surface, 809–810
direct, 810
directions, copying, 823
distribution, 801
for final gather ray, 826
incident, 831
incident directions, 820
indirect, 810, 812, 813, 821, 823
interpolation, 827–833
local definitions, 830, 831
merging, 806
object arrays, 804
organization with heap, 822
path determination, 811
paths, 806, 808
radiance, 810, 813–817
ray direction samples, 810–811
recording, 808
representation, 805
scattering, 811
searching for, 821
shooting, 805–807, 815
storage with kd-tree, 821
storage with local arrays, 806
structure, 804
unordered array of, 823
weights, 805, 811
photorealistic rendering
goal, 4
ray-tracing algorithm and, 4–15
See also rendering
physical (wave) optics, 424
piecewise constant basis functions, 928–930
defined, 928
illustrated, 929
piecewise-constant 1D functions, 647–650
PDF, 647
See also inversion method
piecewise-constant 2D functions, 671–674
illustrated, 674
pinhole cameras
defined, 5
depth of field and, 316
elements, 5
illustrated, 6
simulation, 6
See also cameras
pixels
addressing, 338
coordinates, 349, 532
discrete integer, 409
display, 323–324
display values and, 526
image, 323
in low-discrepancy sampler, 374
measurements, 760
reconstruction filter, 393
sampling range, 341
splat values, 411, 413
understanding, 337–338
weight, 409
planar mapping, 518
plastic material, 486–487
Killeroo rendered with, 486
modeling, 486
parameters, 486
See also materials
point lights, 609–621
goniophotometric diagram, 618–621
intensity, 285
positioning, 610
projection, 942
sampling, 711–712
scene rendering, 610
spotlights, 611–614
texture projection, 614–618
total power emitted, 611
See also light sources; light(s)
point repulsion algorithms, 888
points, 63–65
adding, 64
auxiliary intersection, 507
barycentric coordinate computation for, 147
bounding box, 71, 72
in camera space, 308, 311
candidate, 891, 892, 896
defined, 63
denoting, 63
distance between, 65
Halton sequence, 364
Hammersley sequence, 364
height, in object space, 596
homogeneous, 75
indirect visibility between, 961
integral over, 756
irradiance at, 288, 898, 910
low-discrepancy, 810
optical thickness between, 581
Poisson, 890, 891
Poisson distribution of, 888–896
sampled, 888, 889, 896–901
samples, 325, 333, 962
in sphere, 721
sphere sample, on spherical light source, 721
subtracting, 64
surface, 888–896
transforming, 85–86
translation and, 78
translucent surfaces, 894
weighted sums of, 65
Poisson disk patterns, 378
Poisson distribution, 888–896
Poisson sphere
criterion cases, 889
implementation, 889–890
point computation, 890
in Poisson disk approximation, 888
test, 892
polar coordinates, 661
polar decomposition, 97
polarization, 282
polka dots, 558–560
antialiasing, 559
application illustration, 558
center, 560
declaration, 559
indices computation, 559

polka dots *(continued)*
radius, 560
postaliasing, 332
power distribution, 645–646
power heuristic, 692, 693
prealiasing, 332
precomputed light transport, 925–986
ambient occlusion, 926, 927
basis functions and, 927–932
categories, 927
defined, 925
overview, 925–927
precomputed diffuse transfer, 969–973
precomputed glossy transfer, 974–982
radiance probes, 956–969
spherical harmonics, 932–956
precomputed radiance transfer (PRT)
diffuse, 969–973
glossy, 974–982
rendering with, 970
techniques, 969
prefiltering, 334–335
prime irradiance cache, 788
primitive interface, 184–191
defined, 186
methods, 186–187
primitive subdivision, 208
primitives
adding to voxels, 199
animated, 189–191
as area light sources, 188
bounding boxes, 201
centroids, computing bounds of, 214
geometric, 187–189
groups, rejection of, 192
instanced, 189
lazy refinement, 197
looping over, in voxel, 206, 207
overlapping, kd-tree nodes, 229
partitioning, 213–217
refining, 186–187, 206
splitting based on midpoint centroids, 216
surface scattering inside, 187
surface shader bound to, 477
transformation, 190–191
in voxel, 202
world space bounds, 199
principle of similarity, 901–902
probability density, 44
probability density functions (PDFs), 640
CDF and, 640
discrete, 644
infinite area light, 729–730
Metropolis sampling and, 655
photon-sampling of directions, 825
piecewise-constant 1D functions, 647, 649
power distribution, 645
sampling direction towards spheres, 722
sampling for infinite area light, 724–725
shape sampling, 716–717
spotlight illumination distribution, 712
probes, 1006–1009
data gathering, 1008
defined, 1006
options, 1007–1008
statistics gathered during, 1071
procedural texturing, 544–553
antialiasing, 545, 551
checkerboard, 546–552
defined, 545
fractional Brownian motion for, 561
implications, 545
implicit pattern definition, 553
UV texture, 545–546
See also texture(s)
projected solid angles, 289–290
determination, 289
illustrated, 289
integrals over, 289–290
See also solid angles
projection
general light sources, 942–943
incident radiance functions, 949–950
infinite area lights, 943–949
lat-long map, 944
light sampling, 713
light sources, 941–949
point lights, 942
with Reinmann integration, 944
projective camera models, 305–318
depth of field, 313–318
implementation, 305
orthographic, 306–310
perspective, 310–313
screen-to-raster, 305–306
transformation matrix, 305
pseudo-random number generator, 1002–1003

Q

quadratic equations, 118, 127
quadrics
checkerboard texture applied to, 547
cone, 133–134
cylinder, 124–129
defined, 113
disk, 129–132
hyperboloid, 134–135
paraboloid, 134
polka dots applied to, 558
sphere, 113–124
surface area, 123
UV texture applied to, 546
quadtrees, 1025
quasi Monte Carlo, 685
quaternions, 92–94
addition, 93
along animation path, 95
defined, 92
inner product, 93
interpolation, 94–96
multiplication, 92
representation, 92
rotation matrix, 93
subtraction, 93

R

radiance, 285–286
absorption and, 577–578
along path, 846
camera path, 849
defined, 33, 285
determination, 34
differential, 295
emitted, 293
equation, 285
exitant, 286–287, 754
illustrated, 286
impossible values, 33
in-scattering and, 582
incident, 286–287, 288, 294, 442, 744
out-scattering and, 580
outgoing, accumulating, 817
outgoing, distribution, 753
outgoing, reflected, 453
over set of directions, 288
properties, 285
rays carrying, 35
reflected, 791, 831
surface integrator, 740
total, 45, 579
values, precomputing, 813–817
Whitted integrator evaluation, 42
radiance photons, 810
creating, 814
exitant radiance computation, 813, 824, 826
partitioning, 816
precomputing, 815
process methods, 825

radiance computation for, 816
range computation, 816
scene rendering with, 814
structure, 815
radiance probes, 956–969
bad placement artifacts, 960
bounding box, 959
coefficients, 966
computing at sample points, 962
contribution computation, 963
creating, 956–965
defined, 956
error sources, 956
incident, 963
integrator, 966
reflected lighting computation with, 966
rendering with, 957
using, 965–969
See also precomputed light transport
radiant exitance, 282
radiant flux, 282
defined, 282
differential, 284
illustrated, 283
radiative equilibrium, 913
radiative transfer, 281
radical inverse
computation, 362
defined, 361
positive integers, 363
sequences using, 362
radiometric distribution, 8
radiometric integrals, 288–293
evaluation of, 288
irradiance, 290, 292, 293
over area, 292–293
over projected solid angle, 289–290
over spherical coordinates, 290–292
working with, 288–293
radiometry, 281–288
basic, 281–288
defined, 261, 281
flux, 282, 283
geometric optics, 281
intensity, 284–285
irradiance, 282–284
luminance, 287–288
Maxwell's equations and, 281
radiance, 285–286
radiant exitance, 282
radiative transfer, 281
random number generator, 892, 1003
random polka dots, 558–560
random ray generation, 247
random sampling, 684
random variables
applying functions to, 639
BSDF, 705
canonical uniform, 639
CDF, 639
continuous, 639–640
defined, 638
discrete, 639
independent, 641
inversion method, 643–650
PDF, 640
rejection method, 650–652
sampling, 643–652
uniform, 665
randomized Halton sequences
defined, 366
permutation tables, 367–368
See also Halton sequences
randomness, 637
raster space
computing, 384
defined, 304
sample point, 309
See also coordinate spaces
ray differentials, 68–70
in antialiasing textures, 46
computation for specular reflection, 512
defined, 32
in finding filter regions, 510
offset rays in, 511
reflected, 510
tracking, 511
transmitted, 510
ray propagation, 13–15
defined, 5
participating media, 13
ray tracing
advantage, 318
early example, 12
environment camera, 318–319
high-performance CPU, 994
image projections, 318
photorealistic rendering and, 4–15
recursive, 5, 11–13
single low-level shape representation, 991
strength of, 992
ray-box intersections, 193–195
defined, 193
illustrated, 193, 194
ray-casting function, 752
ray-cylinder intersections, 126–127
ray-disk intersections, 131–132
ray-object intersections, 7–8
acceleration structure, 8
brute-force approach, 7–8
defined, 4
execution time, 192
point, 7
sphere, 123
ray-plane intersections, 507
ray-primitive intersections, 34, 43
information about, 186
light-scattering properties, 187
ray-shape intersections, 112
ray-triangle intersections, 140–145
barycentric coordinates, 140, 142
cost of, 141
distance along ray, 142
ray-voxel stepping algorithm, 203–204
Rayleigh scattering, 584
rays, 66–70
camera simulation and, 6–7
constructing, 68
defined, 66
denoting, 66
direction, 248
"epsilon" value, 248
final gather, 826–827
floating-point weights, 32
illustrated, 67
image location conversion to, 7
marching process, 879
offset, 505
parametric distance to intersection, 186
parametric form, 7, 67
parent, 68
path, vertex generation, 843
random, 247
sampling, 708
shadow, 9, 23
stepping through voxel grid, 203
testing for intersection, 183
time value, 67
transforming, 88
traversal through kd-tree, 241
tree of, 12
volume integration sampling, 881
world space, 117
read for ownership (RFO), 1035
reader-writer mutexes, 1039
real spherical harmonics, 932–933
reconstruction, 389–402
approximation, 324
artifacts, 332
box filter, 394–396
defined, 324
filter functions, 393–402
Gaussian filter, 396–398
ideal, 391

reconstruction *(continued)*
Mitchell filter, 398–400
in pbrt, 391
pixel filter, 392
pixel values, 391
postaliasing, 332
sinc filter, 400–402
success key, 331
triangle filter, 328, 396
recursive ray tracing
defined, 5
illustrated, 13
mirror-reflection direction, 12
tree of rays, 12
See also ray tracing
reduced extinction coefficient, 902
reduced scattering coefficient, 598, 902
reentrancy expectations, 40–41
reference-counted objects, 1010–1012
references
assignment, 1011
as pointers, 1011–1012
reflectance, 430
BRDF, 294–296, 912
conversion process for, 277–278
differential diffuse subsurface, 906
distribution function, 424
estimation, 703–704
Fresnel, 434–439, 905
hemispherical-directional, 430
hemispherical-hemispherical, 430, 704
Lambertian surface, 753
properties, 423
reflected radiance
computation, 791
computation with diffuse PRT, 972
estimating, 831
outgoing, 453
See also radiance
reflection
BRDF, 294–296
BSSRDF, 296–297
categories, 424
computations, 425
coordinate system, 425
diffuse, 424, 425, 449–451
from direct lighting, 848
direction, 12
distribution properties, 423
equation, 296
evaluation with hierarchical integration, 908, 910
glossy, 680
glossy specular, 424, 425
Lambertian, 446–447
mechanisms, 294
Oren-Nayar, 449–451
perfect specular, 424, 425
retro-reflective, 424, 425
specular, 45, 433, 439–441, 702–703
subsurface, 885
surface, 293–297, 739–871
total internal, 439
reflection models, 423–474
basic interface, 428–432
geometric setting, 425–428
implementations, 424
microfacet, 447–460
sources, 423–424
terminology, 424–425
refraction
index of, 432
indices for objects, 434
specular, 433
regular half angle measured BRDFs, 467–470
advantages, 467
defined, 467
regular vertices, 152
rejection method, 650–652
defined, 650
efficiency, 650
example, 651–652
sample generation, 651
sampling, 650
See also random variables
Renderer interface, 24–26
rendering
aliasing sources, 336–337
APIs, 1045–1046
cleanup after, 1071
defined, 1
equation. *See* light transport equation
with hierarchical integration, 908–911
with irradiance cache, 789–791
loop, 17
main loop, 26–35
with measured BRDF data, 462
Metropolis Light Transport (MLT), 857–861
options, 1056–1058
photorealistic, 4–15
with radiance probes, 957
sample pattern effect on, 351
with virtual light sources, 780–784
volume, 873–922
world end and, 1070–1072
reporting, 1005–1006
error, 1005
progress, 1005–1006
retro-reflective reflection
defined, 424
illustrated, 425
RGB color, 273–279
coefficients, 275
conversion, 279
conversion table computation, 276
conversion to SPDs, 276
conversion to XYZ values, 275
device dependent, 412
emission curves, 274
image film, 406
LED/LCD displays, 273
output values, 413
reflectance spectrum conversion, 278
SPD display, 275
spectrum creation, 280
spectrum functions computation, 277
values, 274, 277
See also spectral representation
Riemann integration, 944, 945
right-handed coordinate system, 56, 57
ringing
defined, 398
effect illustration, 950
reducing, 950–951
rotations, 80–83
angle, 81, 83
arbitrary axis, 82–83
Cranley-Patterson, 384
defined, 80
extracting, 98
interpolation, 91, 99
matrix, 93–94
orthogonal matrix, 80
properties, 80
spherical harmonics (SH), 951–956
x axis, 80–81
y axis, 81–82
z axis, 82
Russian roulette, 680–681
applying, 681
defined, 680
in path sampling termination, 766
thresholds, 774
variance and, 681
weights, 681

S

SAH. *See* surface area heuristic
sample placement, 682–686
quasi Monte Carlo, 685
stratified sampling, 682–685
warping samples and distortion, 685–686
sample points
generating, 888

illustrated, 889
octree, 896–901
radiance probe computation, 962
radiance values computation, 897
raster space, 309
sampled spectrum
arithmetic operations, 266
XYZ matching functions computation, 271–272
samplers
adaptive, 385–389
best-candidate, 378–385
current pixel coordinates, 349
Halton, 363–366
implementations, 323, 338, 340
low-discrepancy, 372–378
main rendering loop interaction, 385–386
ray generation report, 341
stratified, 346
samples
(0,2)-sequence, 370
accessing, 344
bootstrapping, 853
difference comparison, 389
discrete pixel, 353
distribution, 25
emitted rays from lights, 777
generation, 349, 350, 351, 353
implementations, 338
integrators, 343, 344, 354, 376
irradiance, 791–797, 898
LHS, 356
light source, 606
linear segments between, 267
maximum number of values, 340
memory allocation, 345–346
MLT, 838–839, 853, 858–860
nonuniform distribution, 335
number returned, 31
photon ray direction, 810–811
point, 325
position, 324
random, 2D mapping, 666
representation, 342–346
shape ids, 388
spacing, 504
SPD, 266, 267
spectral data, 266
storage, 342
stratified, 353
value, 324
vector, 836
warping, 685–686
sampling
1D patterns, 344
2D patterns, 343, 347
3D patterns, 343
adaptive, 334, 341, 385–389
aliasing and, 324, 331–332
anisotropic microfacet model, 699–701
antialiasing and, 332–335
approximation, 324
area lights, 358, 715–722
artifacts, 332
best-candidate, 378–385
Blinn microfacet distribution, 695–699
BSDFs, 705–708
cosine-weighted hemisphere, 668–670
cylinders, 719
directional, 664
directions, 708
disks, 665–668, 718–719
frequency minimum, 332
FresnelBlend class, 701
goniophotometric lights, 713
ideal, 327–331
image interface, 338–346
image synthesis application, 335–336
importance, 679, 688–693, 734
indirect lighting, 776
infinite area lights, 724–730
Latin hypercube, 355
light sources, 708–730
low-discrepancy, 359–378
method comparison, 348
Metropolis, 652–659
Monte Carlo, 441, 481, 630
nonuniform, 334, 391
paths, 763–764
pattern comparison, 346, 347, 379
patterns, 338
phase functions, 731
pixel, 341
point, 333
point lights, 711–712
prealiasing, 332
process, 327, 328
random, 684
random variables, 643
rays, 708
reflection functions, 693–708
rejection, 650–652
shape sets, 722–724
shapes, 716–717
spheres, 720–722
spotlights, 712–713
stratified, 346–359, 682–686
texture, 503–513
theorem, 332
theory, 323–331
triangles, 670–671, 719
uniform, hemisphere, 663–665
volume integration ray, 881
volume scattering, 730–733
sampling rate
computation, 959
defined, 327
low, 332
texture, 503–509, 528
saturation, 14
scalar contribution function, 837
scalars, 56
division, 59
multiplication, 59
reciprocal computation, 59
scale texture, 520–521
antialiasing and, 521
defined, 520
See also texture(s)
scaling
BxDF adapter, 431–432
nonuniform, 79
properties, 79
transformation, 79–80
uniform, 79
vectors, 59–60
scattering
back, 585
defined, 576
equation, 296
heterogeneous, 886
in, 581–583
Mie, 584
out, 579–581
Rayleigh, 584
subsurface, 297, 599–600, 885–913
surface, 5, 10–11
translucent media, 886
volume, 575–602, 730–733
scene definition, 1058–1072
hierarchical graphics state, 1059–1061
light sources, 1064–1065
object instancing, 1069–1070
shapes and volume regions, 1065–1069
surface and material description, 1063–1064
texture and material parameters, 1061–1062
world end and rendering, 1070–1072
scene description
approaches, 1045–1046
defined, 16
initialization and rendering options, 1051–1058
interface, 1045–1076

scene description *(continued)*
new object implementations, 1073
parameter sets, 1047–1051
parsing, 16, 21, 30
scene definition, 1058–1072
state tracking, 1052–1053
transformations, 1053
scenes
complexity, increased, 992–993
geometric object representation, 22–23
instance usage, 1069
light source, 23
representation, 18–24
Schlick phase function, 586–587
defined, 586
plot, 587
See also phase functions
screen space
decomposition, 39
defined, 304
See also coordinate spaces
screen-to-raster projection, 305–306
second fundamental form, 121–122
self-illumination, 913
self-intersections, 111–112
self-shadowing
microfacet models and, 448
Oren-Nayar model and, 449
semi–infinite media, 886
separable approximations, 984
shading aliasing, 336
shading coordinate system, 425, 427
shading geometry, 112–113
computing, 113
with interpolated normals, 112
triangles, 145–148
shading normals, 146, 147
artifacts, 480
bump mapping and, 492
computation, 490
error types resulting from, 481
shadow rays
defined, 9
existence of intersections, 23–24
intersections, 208
virtual light, 782
shadows, 9
shah function
convolution, 329, 330
Fourier transform, 329
shape sets
acceleration structure, 724
defined, 722
PDF value computation, 723
sampling, 722–724
shapes, 107–181
abstraction of, 107
animated, 1065, 1066, 1067, 1068
with animated transformation matrices, 183
API creation, 1073
bounding, 109–110
closed, 103
cone, 133–134
creation, 1066–1067
cylinder, 124–129
definition of, 108
disk, 129–132
hyperboloid, 134
id, 108–109, 388
incorrect self-intersections, 111–112
interface, 107–113, 716
intersection, 110–111
material, 1067
normals, 103
with object instancing, 183
paraboloid, 134
parametric description, 102
as placeholders, 110
pointers, 108
refinement, 110
sampling, 716–717
shading geometry, 112–113
sidedness, 113
sphere, 113–124
surface area, 113
tetrahedron, 148
triangle, 135–148
shared data, updating, 39–40
sidedness, 113
simulation, 423–424
sinc filter, 400–402
finite extent, 400
graphs, 401
infinite support, 392
Lanczos window, 400
ringing, 392
use results, 401
See also filters; reconstruction
single instruction, multiple data (SIMD) processing, 994
single scattering integrator, 882–885
defined, 882
implementation, 882
sample pattern computation, 884
source term computation, 884
visual effect, 883
See also volume integrators
Snell's law, 432
derivation of, 473
for transmitted directions, 45
Sobol' sequence, 372
solid angles
defined, 284
differential, 291, 292, 293
illustrated, 285
intensity and, 284–285
measure relationship, 289
projected, 289–290
solid checkerboard, 552–553
defined, 552
solid textures, 544–553
checkerboard, 552–553
defined, 544
representation problem, 544–545
See also texture(s)
spatial domain, 325
convolution in, 329
equivalent process, 331
frequency space representation, 327
functions, 327
spatial subdivision, 192
spectral matching curves, 270
spectral power distributions (SPDs)
arbitrary, 270, 280
basis function representation, 261
coefficient computation, 263
coefficient representation, 263
computations with, 261
debugging routine, 265
defined, 261
of fluorescent light, 262
linear interpolation, 265
real-world objects, 261
representations, 270–271
resampling, 268
RGB color, 275
samples, 266, 267
spectral representation, 261–266
basis functions, 263
class, 263–266
design advantages, 263
hiding details of, 263
RGB color, 273–279
Spectrum type, 263
value zero, 264
XYZ color, 270–273
spectral synthesis, 560–565
spectrum
clamping, 265
equality/inequality tests, 264
initialization, 264
luminance, 272
sampled, 266–279
shah function, 329
square root, 265

specular reflection, 439–441
BRDF, 439
direction, 441
effect computation, 45
implementation, 439
Killeroo model, 433
modulation, 443
ray differential computation for, 512
texture filtering, 510, 511
vector, 439, 440
See also reflection
specular refraction, 433
specular transmission, 442–446
BTDF for, 442, 443
direction computation, 445–446
effect computation, 45
geometry, 445
modulation, 443
vector, 443
spheres, 113–124
basic setting, 114
bounding, 116, 893
bounding box, 73
center, 115, 720
construction, 115–116
cosine-weighted sampling, 668–670
defined, 113
DifferentialGeometry initialization, 122–123
illustrated, 115
implementation, 115
implicit form, 114
intersection, 116–119
object space, 115
parameterization, 114–115
parametric definition, 120
parametric form, 114, 119
partial, 119–121
partial derivatives of normal vectors, 121–122
Poisson, 888–892
profile curve, 124
radius, 115, 720
ray-object intersection, 123
sample points, 721
sampling, 720–722
second fundamental form, 121–122
spinning, 90–100
stratified uniform distribution of rays over, 689
surface area, 123–124
texture mapping, 114, 115
uniform sampling of, 663–665
See also shapes
spherical coordinates, 661–662
direction vectors and, 290
formula, 319
integrals over, 290–292
spherical harmonics (SH), 932–956
basis, 932–933
coefficient rotation, 955
coefficient vector pointers, 960
coefficients, 933, 941, 954, 956, 976
defined, 932
efficient evaluation, 935–941
environmental map representation, 934
function definitions, 935
function smoothness, 933
incident radiance function projection, 949–950
light source projection, 941–949
local definitions, 936, 939, 940, 968
normalization factor, 932, 941
as orthonormal, 933
point light source projection to, 942
real, 932–933
ringing reduction, 950–951
rotation matrices, 951, 952, 953
rotational invariance, 951
rotations, 951–956
rotations via Euler decomposition, 953–956
sine/cosine terms for rotations, 956
See also precomputed light transport
spherical linear interpolation (SLERP), 94–96
spherical mapping, 515–517
splits
access position, 236
axis selection, 236
edge, 166
preferences, 234
splitting, 681–682
data structures, 1014
defined, 681
integral estimator and, 682
node plane, 242, 243
primitives, 216
surface area heuristic (SAH), 220–221
spotlights, 611–614
angles, 612, 613
defined, 611
falloff computation, 614
lighting distribution, 15
sampling, 712–713
scene rendering, 612
strength computation, 613
See also light(s); point lights
square-to-circle mapping, 686
star discrepancy
bounds, 361
defined, 360
of sequence, 360–361
start-up bias, 655–656
defined, 655
removing, 656
state tracking, 1052–1053
stationary distribution, 653
statistics, 1007–1009
steady state, light, 282
strata
compact, 684
defined, 346, 682
jittering center point, 346
number of, 349
variance, 683
stratified samples
generation, 353, 356
initialization, 354
shifting to pixel coordinates, 353
See also samples
stratified sampling, 346–359, 682–686
1D, 352
2D, 352
area light samples, 358
best-candidate sampling comparison, 382
curse of dimensionality, 350, 684–685
drawbacks, 378, 684–685
example effect, 684
function definitions, 352
Halton sampler comparison, 365
Latin hypercube sampling (LHS) versus, 356
low-discrepancy sampler comparison, 373
next pixel advance, 357
pattern transformation to points, 686
sample generation, 349, 350, 351, 353
sample shifting to pixel coordinates, 353
strata, 346, 682
uniform, over hemisphere, 689
uniform random distribution versus, 684
utility routines, 352
worst-case situation, 355
See also sampling
subdivision, 160–174
applying fixed number of times, 160
on boundary edge, 165
illustrated, 161
levels of, 160
limit surface and output, 170–174
main loop, 161
mesh topology update, 168–170
object, 192

subdivision *(continued)*
primitive, 208
refinement process, 150
rule for edge split, 166
rules, 148, 170
rules application, 160
spatial, 192
steps, 148
tetrahedron, 148
subdivision mesh
boundaries, 152
closed, 152
consistently ordered, 154
control, 148, 149, 150, 154
data structures, 151
extraordinary vertices, 152
faces, 150, 151, 154
in Loop subdivision scheme, 150
manifold, 152, 154
open, 152
representation, 151–160
split edges, 166
topology initialization, 162
topology update, 168–170
triangular face pointers, 153
subdivision surfaces, 148–174
advantages, 150
applied to Killeroo model, 148–149
bounds, 160
convex hull property, 160, 163
definition of, 148
illustrated, 149
Loop, 150
mesh representation, 151–160
methods, 150
smooth, 150
subdivision, 160–174
triangle meshes, 137
See also surfaces
substrate material, 490
subsurface integrator, 887–888
subsurface light transport, 187, 294
subsurface material, 490, 491
subsurface reflectance, 885, 906, 908
subsurface scattering, 885–913
basis, 297
dipole diffusion approximation, 901–908
illustrated, 887
materials, 599
Poisson distribution of sample points, 888–896
properties, 885
properties, setting, 912–913
rendering with hierarchical integration, 908–911
sample point octree, 896–901
supersampling, 551
surface area
cylinders, 129
disks, 132
kd-tree node, 236
shape interaction, 113
spheres, 123–124
triangles, 145
surface area heuristic (SAH), 217–222
computational expense, 217
concept, 218
defined, 217
partitioning primitives with, 219
splitting at bucket boundaries, 220–221
splitting plane selection, 220
See also bounding volume hierarchies (BVHs); kd-tree accelerator
surface integrators
class relationships, 42
interface, 740
outgoing radiance, 740
See also integrators
surface normals. *See* normals
surface reflection, 293–297
BRDF, 294–296
BSSRDF, 296–297
modeling effects, 293
translucent, 294
surface scattering, 10–11
BRDF, 10
BSDF, 11
defined, 5
determination, 10
geometry, 11
inside primitive, 187
surface shader, 424, 477
surfaces
anisotropic, 425
area density of flux arriving/leaving, 282
composition, 423
description, 1063–1064
diffuse, 424, 425
glossy specular, 424, 425
implicit form, 114
isotropic, 424–425
light transport beneath, 297
limit, 148
microfacets, 447
Möbius strip, 154
parametric form, 114
perfect specular, 424, 425
properties, 424
quadrics, 113
retro-reflective, 424, 425
smooth, 432
subdivision, 148–174
translucent, 894

T

tangents, 136, 137, 142
across, 172
computing for boundary vertices, 172–173
computing for interior vertices, 172
shading, 147
transverse, 172–173
task queue, 36
task system, 36, 1040–1041
tasks
adding to work queue, 1041
class synchronization for, 31
cleanup, 1041
execution of, 28
functions, 30
implementation, 1041
number determination, 28–29
run time distribution, 29
shared data updating, 39–40
variables, 31
"teapot in a stadium" problem, 196
testing
aggregate implementation, 246–247
ray intersection, 183
voxel overlap, 200
tetrahedron subdivision, 148
texture coordinates, 513–519
2D (u, v) mapping, 514–515
3D mapping, 518–519
checkerboard textures, 546
cylindrical mapping, 517
differential change, 519
differentials for sphere, 517
discontinuity, 516
floating-point, 538
planar mapping, 518
selection methods, 513
spherical mapping, 515–517
texture filtering
elliptical weighted average, 539
problem, 509
for specular reflection, 510, 511
triagular filter, 536
texture maps
AO values in, 927
defined, 523
filtering, 529
procedural texturing and, 545
See also image textures

texture projection lights, 614–618
defined, 614
directions behind, discarding, 617
mip map creation, 616
projection matrix initialization, 616
scene rendering, 615
setting illustration, 615
See also light(s); point lights
texture space, 542
texture(s), 501–573
aliasing, 501, 502
alpha mask, 136
antialiasing, 503–513
bilinear, 522–523
bumpy, 565–566
caching, 524–527
checkerboard, 348, 514, 546–552
constant, 520
displacement, 494, 495
functions, 501
functions, filtering, 509–510
as high-frequency variation source, 501
ideal resampling, 509
image, 523–544
implementations, 514
inline functions, 563
interface, 519–520
marble, 567–569
mix, 521–522
noise, 553–569
parameters, 1061–1062
in `pbrt`, 501
polka dot, 558–560
procedural, 544–553
sampling, 503–513
sampling rate, 503–509, 528
scale, 520–521
solid, 544–553
solid checkerboard, 552–553
UV, 545–546
windy waves, 566–567
wrinkled, 565–566
three-dimensional bounding boxes, 70–74
throughput processors, 993–996
Torrance-Sparrow model, 43, 452–455
derivation of, 452, 453
Fresnel term, 455
geometric attenuation, 453, 455
half-angle vector, 452
implementation, 454
Killeroo model, 456
reflected outgoing radiance, 453
terms evaluation, 455
for two outgoing directions, 468
See also microfacet models
total internal reflection, 439
transfer function, 971, 977
transfer matrix, 975–977
BSDF, 978
SH radiance, 981
transformations, 74–85
affine, 97
animating, 90–100
applying, 85–89
array, 1054
benefits, 74
of bounding boxes, 88
composite matrices, 96–97
composition of, 88–89, 97
continuous, 74
coordinate system handedness and, 89
decomposition, 97
defined, 74
between distributions, 660–662
of the frame, 74
functions, 1054
homogeneous coordinates, 75–76
identity, 76
invertible, 74
keyframe, 90
linear, 74
look-at, 84–85
matrix, 77, 88
multidimensional, 662–674
multiple dimensions, 661
multiplying, 89
of normals, 86–88
from one frame to another, 74
one-to-one, 74
operations, 76–77
pointers, 1066
of points, 85–86
polar, 661
primitive, 190–191
of rays, 88
rotation, 80–83
scaling, 79–80
scene description, 1053–1056
translations, 77–79
of vectors, 86
world-to-object, 123
translations, 77–79
defined, 77
extracting, 97
interpolation, 91
matrix equation, 79
in matrix form, 78
points and, 78
properties, 78
in 2D, 78
translucent material, 483, 490, 894
transmission. *See* specular transmission
transmittance, 13
beam, 580–581
Fresnel, 909
transverse tangents, 172–173
traversal
bounding volume hierarchies (BVHs), 224–227
grid accelerator, 202–208
kd-tree accelerator, 240–245
triangle filter
defined, 396
graph, 395
See also filters; reconstruction
triangle meshes
bound computation, 138
coordinate assignment, 137
defined, 135
illustrated, 136
object space bound, 138
shape transformation, 137
smooth-looking, 171
subdivision surfaces, 137
tangent vectors, 142
vertices, 152
triangles, 135–148
dual role, 136–137
intersection, 140–145
only, as design alternative, 991–992
parametric coordinates, 142
partial derivatives, 142, 143
sampling, 670–671, 719
as set of points, 142
shading geometry, 145–148
storage reduction, 139
surface area, 145
vertices, 139–140
world space bounding boxes, 109, 139
See also shapes
triple product integral, 984
tristimulus theory, 270, 271
turbulence, 563–565
defined, 563
graphs of, 564
implementation, 563
noise octave computation, 565
See also noise

U

uber material, 490
unified resampling filters, 539
uniform scaling, 79
unit vectors, 61, 62

utilities, 999–1043
communication with user, 1004–1006
functions, 1000–1003
image file input/output, 1003–1004
kd-trees, 1027–1033
main include file, 999–1000
mathematical routines, 1020–1021
memory management, 1009–1020
octrees, 1022–1027
parallelism, 1033–1041
probes and statistics, 1006–1009
pseudo-random number generator, 1002–1003
UV texture, 545–546
application illustration, 546
defined, 545

V

valence, 152
value noise, 554
van der Corput sequences, 362, 368, 372
variable stack allocation, 1009
variance reduction
bias, 686–687
importance sampling, 688–693
multiple importance sampling, 690–693
Russian roulette, 680–681
sample placement, 682–686
sampling reflection functions, 693–708
splitting, 681–682
variance
of functions, 640, 641
Monte Carlo ray tracing, 679
random sampling and, 684
strata, 683
sum of, 641
varying-density volumes, 591–596
3D grids, 592–594
exponential density, 594–596
protected data, 592
vectors, 57–63
addition, 58
arithmetic, 58–59
basis, 55–56, 83
coordinate system from, 63
cross product, 61–62
difference, 468
direction, 57, 290
division, 59
dot product, 60–61, 556
gradient, 556
half-angle, 452, 697, 700
homogeneous, 75
memory area, 1015
multiplication, 59
normal, 65–66, 86–88
normalization, 62–63
obtaining, 64
rotation around arbitrary axis, 82
sample, 836
scaling, 59–60
specular reflection, 439, 440
specular transmission, 443
subtraction, 58–59
tangent, 136, 137, 142, 147
transforming, 86
unary negation operator, 60
unit, 61, 62
"up," 85
vertices
boundary, 156, 162
boundary, determining, 157
boundary, valance computation, 158–159
child faces, 170
edge computation, 166
edges rule application, 167–168
even, 162
extraordinary, 152, 157, 162
indices of, 156, 170
initialization, 157, 158, 162, 167
interior, 162
limit surface position, 171
local support, 163
neighboring faces, 154
new position computation, 162–168
next face, 157, 158
nonboundary, valance computation, 158
numbering scheme, 168
odd, 162, 168
one-ring rule, 162, 172
parent, 169
previous face, 157
regular, 152, 162
sorting, 155
split edges computation, 161
tangents computation, 171
updating values, 170
valance, 152, 157, 158
weights, 163, 171
viewing volume, 5
virtual light sources
creating, 775–780
indirect illumination with, 780
path termination, 780
at ray intersection point, 779
rendering with, 780–784
sample ray leaving, 778
shadow ray, 782
tentative contribution computation, 782
weak singularity effect, 781
visibility, 9–10
defined, 5
indirect, between two points, 961
indirect, efficiently computing, 962
visibility testing, 608–609
defined, 608
volume aggregates, 596–598
defined, 596
implementations, 597
reasons for, 596
volume integrators, 876–885
emission-only, 876–882
interface, 876
multiple scattering, 886
single scattering, 882–885
stepping, 880–881
uniform sampling, 730
volume rendering, 873–922
emission-only integrator, 876–882
equation of transfer, 873–875
single scattering integrator, 882–885
subsurface scattering, 885–913
volume integrator interface, 876
volume scattering, 575–602, 730–733
absorption, 575, 576–578
BSSRDF, 598–600
definitions, 585, 597, 598, 733, 913
emission, 575, 578–579
homogeneous properties, 576
homogeneous volumes, 589–591
illustrated, 576
in-scattering, 581–583
inhomogeneous properties, 576
interface, 587–588
local definitions, 912
optical thickness computation, 731–733
out-scattering, 579–581
participating media, 575
phase functions, 583–587, 731
processes, 575–583
properties, setting, 912–913
sampling, 730–733
varying-density volumes, 591–596
volume aggregates, 596–598
volumes
homogeneous, 589–591
light transport equation, 13
regions, 730, 1072
varying-density, 591–596
volumetric lighting. *See* radiance probes
volumetric primitives, 23

voxel grids
 resolution, 198
 size, 198
 stepping axis, 207
 stepping ray through, 203, 205
voxels
 adding primitives to, 199
 address computation, 204
 allocating, 199
 defined, 195
 function of, 195
 intersection checking in, 205
 looping over primitives in, 206, 207
 number of, 197, 198, 199
 number of primitives in, 206
 overlap testing, 200
 primitive storage location determination, 201
 primitives in, 202
 stepping direction, 205
 stepping logic, 204
 See also grid accelerator

W

warping samples, 685–686
weak singularity, 781
Web site, this book, 49
Weingarten equations, 121, 128
Whitted integrator, 41–47
 functions, 41
 geometric setting, 44
 radiance evaluation, 42
 scattered light and, 45
windowed sinc filter, 400–402
windy waves, 566–569
 defined, 566–567
 evaluation function, 567
 See also noise
world end, 1070–1072
world space, 56
 AABB, 587
 bounding boxes, 109, 139
 defined, 303
 primitive transformation to, 191
 rays, 117
 widths, 199
wrinkled textures, 565–566

X

XYZ color, 270–273
 conversion, 279
 converting RGB to, 275
 image film, 406
 integrals, 280
 matching curves, 271
 SPD representation, 270, 271
 value computation, 271
 values, 271
 y coordinate, 272
 See also spectral representation

This page intentionally left blank

Physically Based Rendering

FROM THEORY TO IMPLEMENTATION

This book was typeset with TEX, using the ZzTEX macro package on the Microsoft Windows XP platform. The main body of the text is set in Minion at 9.5/12, and the margin indices are set in Bitstream Letter Gothic 12 Pitch at 5.5/7. Chapter titles are set in East Bloc ICG Open and Univers Black. Cholla Sans Bold is used for other display headings.

The manuscript for this book is written in `pyweb`, a literate programming markup format of the authors' own design. This input format is based heavily on the `noweb` system developed by Norman Ramsey. The `pyweb` scripts simultaneously generate the TEX input for the book as well as the source code of the `pbrt` system.

In addition, these scripts semi-automatically generate the code identifier cross-references that appears in the margin indices. Wherever possible, these indices are produced automatically by parsing the source code itself. Otherwise, usage and definition locations are marked explicitly in the `pyweb` input, and these special marks are removed before either the book or the code is generated. These scripts were originally written by the authors, and subsequently rewritten by Paul Anagnostopoulos in Gossip to integrate into the ZzTEX package.

Overall, the book comprises over 90,000 lines of `pyweb` input, or nearly 5 megabytes of text. The cover image, example renderings, and chapter images were generated by `pbrt`, the software that is described in this book.

Index of Notation

p	Point in 3D space	56
$\mathbf{v}$	Direction vector in 3D space	56
p_x, $\mathbf{v}_x$	x component of point / vector	57
$\mathbf{v} \cdot \mathbf{w}$	Dot product of vectors $\mathbf{v}$ and $\mathbf{w}$	60
$\|\|\mathbf{v}\|\|$	Length of vector $\mathbf{v}$	61
$\mathbf{v} \times \mathbf{w}$	Cross product of vectors $\mathbf{v}$ and $\mathbf{w}$	61
$\hat{\mathbf{v}}$	Normalized vector $\mathbf{v}$	62
$\mathbf{T}$, $\mathbf{M}$	Linear transformation (4×4 matrix)	74
$\mathbf{q}$	Quaternion	92
(u, v)	2D parametric coordinates of point on surface	101
λ	Wavelength of light	261
$X(\lambda)/Y(\lambda)/Z(\lambda)$	XYZ spectral matching curves	270
Φ	Radiant flux (power)	282
E	Irradiance (flux area density)	282
M	Radiant exitance (flux area density)	282
ω	Normalized vector on the unit sphere	284
I	Radiant intensity (flux density per solid angle)	285
L	Radiance (flux density per area per solid angle)	285
$L_\mathrm{i}(\mathrm{p}, \omega)/L_\mathrm{o}(\mathrm{p}, \omega)$	Incident and exitant radiance functions	286
Y	Luminance	287
Ω	Set of directions on the unit sphere	288
$\mathcal{H}^2(\mathbf{n})$	Hemisphere about surface normal $\mathbf{n}$	288
$f_\mathrm{r}(\omega_\mathrm{i}, \omega_\mathrm{o})$	BRDF (bidirectional reflectance distribution function)	295
$f(\omega_\mathrm{o}, \omega_\mathrm{i})$	BSDF (bidirectional scattering distribution function)	296
$\mathbb{S}^2$	Sphere of directions	296
$S(\mathrm{p}_\mathrm{o}, \omega_\mathrm{o}, \mathrm{p}_\mathrm{i}, \omega_\mathrm{i})$	BSSRDF (bidirectional scattering-surface reflectance distribution function)	296
$F(\omega)$	Frequency-space representation of function $f(x)$	326
$\delta(x)$	Dirac delta distribution	327
$\mathrm{III}_T(x)$	Shah function with period T	327
$\otimes$	Convolution operation	327
$\Pi_T(x)$	Box function with period T	329
$\Phi_b(n)$	Radical inverse function, base b	361
$\rho_\mathrm{hd}(\omega)$	Hemispherical-directional reflectance	430
ρ_hh	Hemispherical-hemispherical reflectance	430
η_i, η_o	Index of refraction (incident and outgoing media)	432
F_r	Fresnel reflectance	435
k	Absorption coefficient	435
ω_h	Half-angle vector	452
$D(\omega_\mathrm{h})$	Microfacet distribution function	452

Index of Notation

$G(\omega_o, \omega_i)$	Microfacet geometric attenuation term	454
(s, t)	2D texture coordinates	514
σ_a	Absorption cross section	576
$L_{ve}(p, \omega)$	Emission in a participating medium	578
σ_s	Scattering coefficient	579
σ_t	Attenuation (absorption) coefficent	580
$T_r(p \to p')$	Beam transmittance	580
$\tau(p \to p')$	Optical thickness	581
$p(\omega \to \omega')$	Phase function	582
$S(p, \omega)$	Source term	582
X, Y	Random variables	638
$P(x)$	Cumulative distribution function (CDF)	639
ξ	Canonical uniform random variable	639
$p(x)$	Probability density function (PDF)	640
$E_p[f(x)]$	Expected value of a function over distribution p	640
$E[f(x)]$	Expected value of a function over uniform distribution	640
$V[f(x)]$	Variance	641
$T(X \to X')$	Transition function	653
$a(X \to X')$	Acceptance probability	653
J_T	Jacobian of multidimensional function	661
$p(y\|x)$	Conditional density function	662
$\epsilon[F]$	Efficiency of Monte Carlo estimator	679
$t(p, \omega)$	Ray casting function	752
$L(p' \to p)$	Exitant radiance from p' to p	754
$f(p'' \to p' \to p)$	Three-point form of the BSDF	754
$G(p \leftrightarrow p')$	Geometric coupling term	754
$V(p \leftrightarrow p')$	Visibility term	754
$P(\bar{p}_i)$	Contribution of $i + 1$ vertex path	756
$T(\bar{p}_i)$	Throughput of $i + 1$ vertex path	757
$W(p, \omega)$	Importance function	760
α	Particle weight	776, 798
σ_s'	Reduced scattering coefficient	902
σ_t'	Reduced extinction coefficient	902
$\phi(p)$	Fluence	904
F_{dr}	Diffuse Fresnel reflectance	905
$R_d(p_i, p_o)$	Diffuse subsurface reflectance	906
$Y_l^m(\theta, \phi)$, $Y_l^m(\omega)$	Spherical harmonic basis function	932
$T(p, \omega)$	Transfer function	971